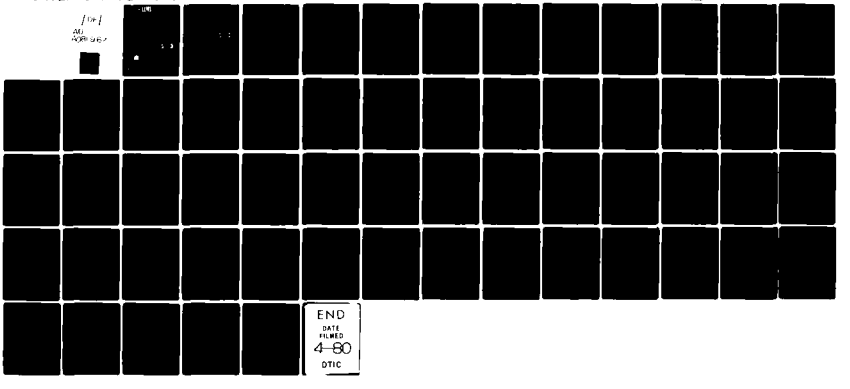
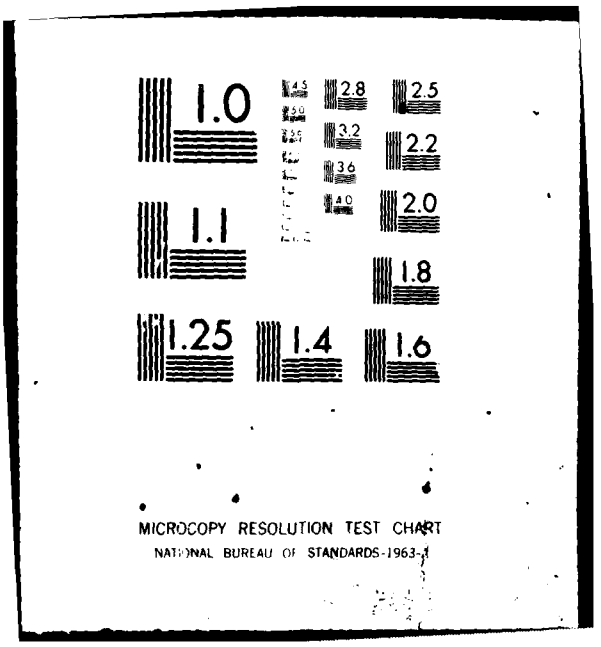
AD-A081 967 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
THE STRUCTURE OF PARALLEL ALGORITHMS.(U)
AUG 79 H T KUNG N00014-76-C-0370
UNCLASSIFIED CMU-CS-79-143 NL



END
DATE FILMED
4-80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

THE STRUCTURE OF PARALLEL ALGORITHMS

H. T. Kung

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

August 1979

See 1473 in back.

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-79-143

THE STRUCTURE OF PARALLEL ALGORITHMS

H. T. Kung

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

August 1979

(To appear in the forthcoming Advances in Computers, Volume 19, Academic Press)

This research is supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.

ABSTRACT

The purpose of this article is to create a general framework for the study of parallel algorithms. A taxonomy of parallel algorithms, based on their relations to parallel computer architectures, is introduced. Examples of parallel algorithms for many architectures are given; they include algorithms for SIMD array processors, for MIMD multiprocessors, and for direct chip implementations. By presenting these algorithms in a single place, issues and techniques in designing algorithms for various types of parallel architectures are discussed and compared.

Table of Contents

1. Introduction

There is a large body of literature on parallel algorithms. Parallel algorithms have been studied since the early sixties (see the survey [Miranker 71]), although at that time no parallel computers had been constructed. It has always been the case that many researchers find designing parallel algorithms fascinating and challenging, regardless of whether or not their algorithms will be used in practice. Increasing interests in parallel algorithms are created by the emergence of large scale parallel computers in the past decade. As a result, a variety of algorithms have been designed for various parallel computer architectures. For surveys of parallel architectures and parallel algorithms see [Anderson and Jensen 75, Stone 75, Kung 76, Enslow 77, Kuck 77, Ramamoorthy and Li 77, Sameh 77, Heller 78, Kuck 78]. The recent advent of large scale integration technology has further stimulated interests in parallel algorithms. Algorithms have been designed for direct chip implementation (see, e.g., [Kung 79]). Hence there is a vast amount of parallel algorithms known today, designed from many different viewpoints.

This article presents many examples of parallel algorithms and studies them under a uniform framework. In Section 2 we identify three important attributes of a parallel algorithm and classify parallel algorithms in terms of these attributes. Our classification of parallel algorithms corresponds naturally to that of parallel architectures. Algorithms for synchronous parallel computers are considered in Section 3, where examples of algorithms using various communication geometries are presented. Section 4 considers algorithms for asynchronous parallel computers. In that section, we discuss a number of techniques to deal with the difficulties arising from the asynchronous behavior of computation, and our examples are mainly drawn from results in concurrent database systems. Section 5 contains some concluding remarks.

The author hopes that by presenting parallel algorithms of many different types in a single place, this article can be useful to readers who wish to understand the basic issues and techniques in designing parallel algorithms for various architectures. The article can be useful as well to readers who wish to know what parallel algorithms are available, in order to decide on the best way to design or choose a parallel architecture.

2. The Space of Parallel Algorithms: Taxonomy and Relation to Parallel Architectures

2.1 Introduction

We view a parallel algorithm as a collection of independent task modules which can be executed in parallel and which communicate with each other during the execution of the algorithm. In Section 2.2, we identify three orthogonal dimensions of the space of parallel algorithms: concurrency control, module granularity, and communication geometry. Along each dimension, we illustrate some important positions that parallel algorithms can assume, but no attempt will be made to list all possible positions. In Section 2.3, we characterize parallel algorithms that correspond to three important parallel architectures along the concurrency control and module granularity dimensions. In Section 2.4, this characterization together with the third dimension -- communication geometry -- forms a taxonomy for parallel algorithms. Our taxonomy is crude and is by no means meant to be complete. The main purpose of introducing it here is to provide a framework for later discussions in this paper. We hope that future work on the taxonomy will make it possible to unambiguously classify parallel algorithms at a conceptual level, and to relate each parallel algorithm to those parallel architectures to which it naturally corresponds.

2.2 The Three Dimensions of the Space of Parallel Algorithms

Concurrency Control

In a parallel algorithm, because more than one task module can be executed at a time, concurrency control is needed to ensure the correctness of the concurrent execution. The concurrency control enforces desired interactions among task modules so that the overall execution of the parallel algorithm will be correct. The leaves of the tree in Fig. 2-1 represent the space of concurrency controls which can be used in parallel algorithms. For example, the left most leaf represents the concurrency control of an algorithm whose task modules execute in lock-step the same code broadcast by the central control, while the second left most leaf represents a synchronous distributed control achieved by simple local control mechanisms.

Module Granularity

The module granularity of a parallel algorithm refers to the maximal amount of computation a typical task module can do before having to communicate with other modules. The module granularity of a parallel algorithm reflects whether or not the algorithm tends to be communication intensive. For example, a parallel algorithm with a small module granularity

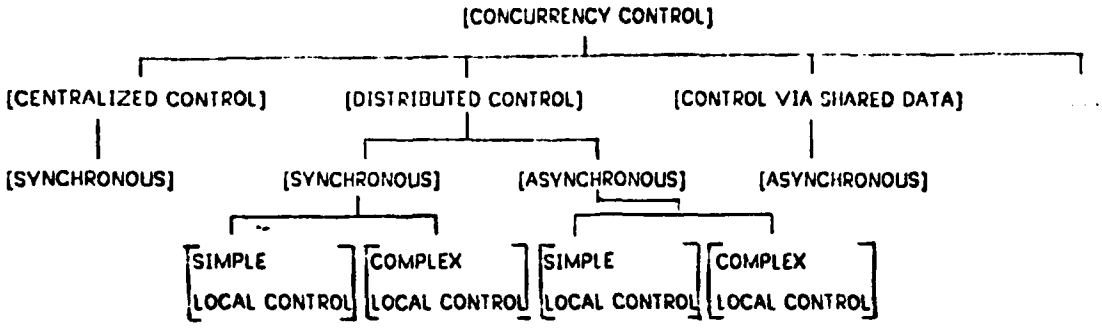


Figure 2-1: A classification of concurrency controls of parallel algorithms - leaves of the tree representing various types of concurrency controls.

will require frequent intermodule communication. In this case, for efficiency reasons it may be desirable to provide proper data paths in hardware to facilitate the communication. For the purpose of this paper, we shall classify module granularities of parallel algorithms into only three groups. See Fig. 2-2.

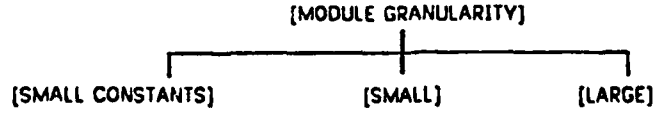


Figure 2-2: A classification of module granularities of parallel algorithms.

Communication Geometry

Suppose that task modules of a parallel algorithm are connected to represent intermodule communication. Then a geometric layout of the resulting network is referred to as the communication geometry of the algorithm. The leaves of the tree in Fig. 2-3 represent the space of communication geometries. For example, leaf HEXAGONAL represents communication geometries that correspond to regular 2-dimensional hexagonal arrays (see Fig. 3-9 (b)).

2.3 Matching Parallel Algorithms with Parallel Architectures

It is straightforward for one to assess the matching between parallel algorithms and parallel architectures along the communication geometry dimension. Here we discuss the less obvious matching along the other two dimensions: concurrency control and module

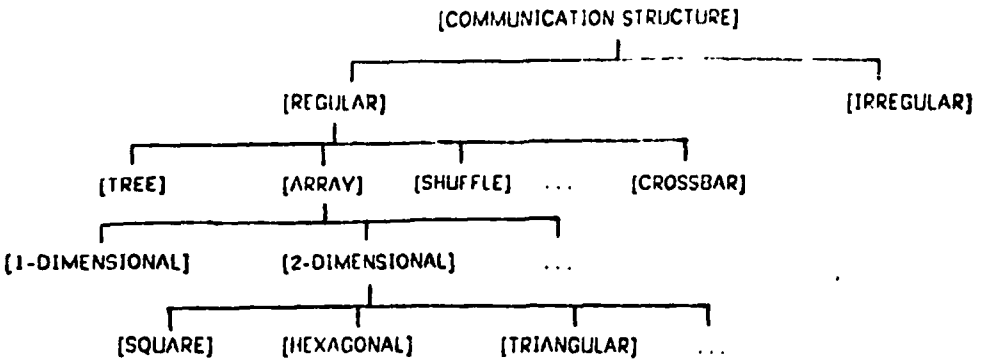


Figure 2-3: A classification of communication geometry of parallel algorithms - leaves of the tree representing various types of communication structures.

granularity. We consider three architectures and their matching algorithms that are relevant to our discussions in Sections 3 and 4.

SIMD and MIMD Machines and Algorithms

The notion of single-instruction stream multiple-data stream (SIMD) and multiple-instruction stream multiple-data stream (MIMD) parallel computers [Flynn 66] is often used in the literature for classifying parallel computers. With a SIMD machine such as ILLIAC IV [Barnes et al. 68], one stream of instructions issued by the central control unit controls all the processors, each operating upon its own memory synchronously. With a MIMD machine such as C.mmp [Wulf and Bell 72], Cm* [Fuller, et al. 77], or Pluribus [Heart et al. 73], the processors have independent instruction counters, and operate asynchronously on shared memories. SIMD machines correspond to synchronous lock-step algorithms that require central controls, whereas MIMD machines correspond to asynchronous algorithms with relatively large granularities [Kung 76]. Algorithms that match with SIMD and MIMD machines are called SIMD and MIMD algorithms, respectively. See Fig. 2-4.

Systolic Machines and Algorithms

Developments in microelectronics have revolutionized computer design. Large Scale Integration (LSI) technology has increased the number and complexity of components that can fit on a chip. In fact, component density has been doubling every one-to-two years for more than a decade. Today a single chip can contain hundreds of thousands of devices. As a result, machines-on-a-chip have emerged; these machines can be used as special purpose devices attached to a conventional computer. "Systolic machines" represent one class of such machines that have regular structures. Intuitively a systolic machine is a network of simple

	CONCURRENCY CONTROL	MODULE GRANULARITY
SYSTOLIC	DISTRIBUTED CONTROL ACHIEVED BY SIMPLE LOCAL CONTROL MECHANISMS, LOCK STEP IN GENERAL	SMALL CONSTANTS
SIMD	CENTRALIZED, LOCK STEP (SINGLE-INSTRUCTION MULTIPLE-DATA STREAM)	SMALL CONSTANTS, SMALL, OR LARGE
MIMD	VIA SHARED DATA, ASYNCHRONOUS (MULTIPLE-INSTRUCTION MULTIPLE-DATA STREAM)	LARGE

Figure 2-4: Characterizations of parallel algorithms that match with systolic, SIMD, AND MIMD machines, along the concurrency control and module granularity dimensions.

and primitive processors that circulate data in a regular fashion [Kung and Leiserson 79]. The word "systole" was borrowed from physiologists who use it to refer to the rhythmically recurrent contractions of the heart and arteries which pulse blood through the body. For a systolic machine, the function of a processor is analogous to that of the heart. Each processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network. At every processor the control for communication and computation is very simple, and the storage space is only a small constant, independent of the size of the network. For a low cost and high performance chip implementation, it is crucial that the geometry of the communication paths in a systolic machine be simple and regular. The geometric problem will be treated in detail in Section 3. Systolic machines correspond to synchronous algorithms that use distributed control achieved by simple local control mechanisms and that have (small) constant module granularities. Algorithms that match with systolic machines are called systolic algorithms. See Fig. 2-4.

2.4 A Taxonomy for Parallel Algorithms

Let {CONCURRENCY CONTROLS}, {MODULE GRANULARITIES}, and {COMMUNICATION GEOMETRIES} be the sets of leaves in Fig. 2-1, 2-2, and 2-3, respectively. Then the cross product {CONCURRENCY CONTROLS}x{MODULE GRANULARITIES}x{COMMUNICATION GEOMETRIES} represents the space of parallel algorithms. One could give a taxonomy for parallel algorithms which classifies algorithms in terms of their positions in this three-dimensional space, but the space is seen to be large and contains quite a few uninteresting cases. We therefore restrict ourselves to a small subspace which nevertheless contains, we believe, most of the interesting and significant parallel algorithms. This subspace is the cross product {SYSTOLIC, SIMD, MIMD}x{COMMUNICATION GEOMETRIES}, where SYSTOLIC, SIMD AND MIMD are three particular positions in the space {CONCURRENCY CONTROLS}x{MODULE GRANULARITIES} that represent systolic, SIMD, AND MIMD algorithms, respectively (c.f. Fig. 2-4).

We name algorithms in {SYSTOLIC, SIMD, MIMD}x{COMMUNICATION GEOMETRIES} in a natural way. For example, an algorithm is called a systolic algorithm using a hexagonal array, if it is systolic and its communication geometry is a hexagonal array.

Generally speaking, among the three types of algorithms (SYSTOLIC, SIMD and MIMD), systolic algorithms are most structured and MIMD algorithms are least structured. For a systolic algorithm, task modules are simple and interactions among them are frequent. The situation is reversed for MIMD algorithms. Systolic algorithms are designed for direct hardware implementations, while MIMD algorithms are designed for executions on general purpose multiprocessors. SIMD algorithms may be seen as lying between the other two types of algorithms. Using the central control, SIMD algorithms can broadcast parameters and handle exceptions rather easily. These reasons make SIMD algorithms attractive in some cases.

In summation, along the concurrency control and module granularity dimensions we have classified parallel algorithms into three classes: SYSTOLIC, SIMD, and MIMD. Each class of algorithms can further adopt various communication geometries. Figure 2-5 presents examples in the space {SYSTOLIC, SIMD, MIMD}x{COMMUNICATION GEOMETRIES}. Most of these parallel algorithms will be discussed in the rest of the paper. Systolic and SIMD algorithms will be treaded in Section 3, whereas MIMD algorithms will be studied in Section 4.

ALGORITHM TYPES	EXAMPLES
SYSTOLIC ALGORITHMS USING	
1-DIM LINEAR ARRAYS	REAL-TIME FIR-FILTERING, DISCRETE FOURIER TRANSFORM (DFT), CONVOLUTION, MATRIX-VECTOR MULTIPLICATION, RECURRENCE EVALUATION, SOLUTION OF TRIANGULAR LINEAR SYSTEMS, CARRY PIPELINING, SORTING, PRIORITY QUEUE, CARTESIAN PRODUCT, PIPELINE ARITHMETIC UNITS
2-DIM SQUARE ARRAYS	PATTERN MATCHING, GRAPH ALGORITHMS INVOLVING ADJACENCY MATRICES, DYNAMIC PROGRAMMING FOR OPTIMAL PARENTHESIZATION
2-DIM HEXAGONAL ARRAYS	MATRIX PROBLEMS (MATRIX MULTIPLICATION, LU-DECOMPOSITION BY GAUSSIAN ELIMINATION WITHOUT PIVOTING, QR-FACTORIZATION), TRANSITIVE CLOSURE, DFT, RELATIONAL DATABASE OPERATIONS
TREES	SEARCHING ALGORITHMS (QUERIES ON NEAREST NEIGHBOR, RANK, ETC., SYSTOLIC SEARCH TREE), PARALLEL FUNCTION EVALUATION, RECURRENCE EVALUATION
SHUFFLE-EXCHANGE	FAST FOURIER TRANSFORM, BITONIC SORT
SIMD ALGORITHMS	NUMERICAL RELAXATION FOR PARTIAL DIFFERENTIAL EQUATIONS OR IMAGE PROCESSING, GAUSSIAN ELIMINATION WITH PIVOTING, MERGE SORT. (IN GENERAL, CORRESPONDING TO EACH SYSTOLIC ALGORITHM THERE IS A SIMD ALGORITHM CONSISTING OF TASK MODULES WITH LARGER GRANULARITIES.)
MIMD ALGORITHMS	CONCURRENT DATABASE ALGORITHMS (CONCURRENT ACCESSES TO B-TREES OR BINARY SEARCH TREES, CONCURRENT DATABASE REORGANIZATION - GARBAGE COLLECTION), CHAOTIC RELAXATION, DYNAMIC SCHEDULING ALGORITHMS, ALGORITHMS WITH LARGE MODULE GRANULARITIES

Figure 2-5: Examples in the parallel algorithm space.

3. Algorithms for Synchronous Parallel Computers

3.1 Introduction

We consider in this section parallel algorithms for synchronous parallel computers, which include systolic and SIMD machines described in Section 2. These algorithms will be classified, to first order at least, according to their communication geometries. Results in this section should provide useful insights into the problem of selecting interconnection networks for systolic or SIMD machines.

As mentioned in Section 2.3, the existence of a cost effective chip implementation of a systolic algorithm in LSI technology depends crucially on the communication geometry of the algorithm. It is highly desirable that communication geometries be simple and regular. Such structures lead to cheap implementations and high densities. In turn, high density implies both high performance and low overhead for support components. For more discussions on this matter, see [Sutherland and Mead 77, Foster and Kung 79]. In this section, special attention will be paid to those structures which are simple and regular.

One of the main concerns in the design and verification of synchronous algorithms defined on networks is to ensure that required data items will reach the right places at the right times to interact with each other. For this reason, we shall often illustrate algorithms by their data flow diagrams. For systolic machines, further attention is needed to ensure that the execution of a task module requires only a small constant amount of time and space. We assume throughout the section that it takes a unit of time to send a unit of data from a processor to any of its topological neighbors. (See discussions in Section 3.4 for the rationale of this assumption for a case involving wires of different lengths.) Under this assumption, we shall show that many problems which require nonlinear (e.g., $O(n \log n)$, $O(n^2)$, or $O(n^3)$) times on uniprocessors can be solved in linear times on systolic machines with enough processors. Algorithms for systolic machines can run on corresponding SIMD machines with similar underlying interconnection structures without losing efficiency, but not vice versa. The unique capabilities of SIMD machines for broadcasting data and instruction codes, and for storing a relatively large amount of data local to each processor can be crucial to the efficiency of some algorithms. Algorithms presented in this section are in general suitable for systolic machines, unless stated otherwise.

3.2 Algorithms Using One-dimensional Linear Arrays

One-dimensional linear arrays (Fig. 3-1) represent the simplest and also the most fundamental geometry for connecting processors. Shift-resisters can implement linear arrays directly. Surprisingly enough, for a large number of important algorithms this simple structure is all that is needed for communication.

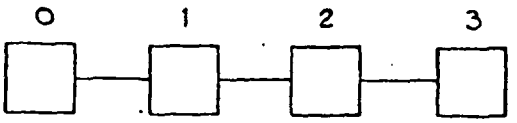


Figure 3-1: A one-dimensional linear array.

In the following, we give four algorithms using linear arrays. The first algorithm concerning odd-even transposition sort is perhaps the most well-known algorithm using a linear processor array (see, for example, [Knuth 73] and [Mukhopadhyay and Ichikawa 72]). The latter three algorithms demonstrate an important way of using linear arrays. That is, a linear array can be viewed as a pipe and thus is natural for pipeline computations. Depending on the algorithm, data may flow in only one direction or in both directions simultaneously. We show that two-way pipelining is a simple and powerful construct for realizing complex computations. Following the discussions of the four algorithms, we mention the use of linear pipelines in the implementation of arithmetic operations.

For ease in describing these algorithms, we shall number the processors from left to right by integers 0, 1, . . . , as in Fig. 3-1.

Odd-Even Transposition Sort

Given n keys stored in a linear array of processors, one key in each processor, the problem is to sort them in ascending order. The problem can be solved in n steps by using the odd-even transposition sort. Odd and even numbered processors are activated alternately. Assume that the even numbered processors are activated first. In each cycle, the following comparison-exchange operations take place: the key in every activated processor is first compared with the key in its right hand neighboring processor, and then the smaller one is stored in the activated processor. Within n cycles, the keys will be sorted in the array (see Fig. 3-2).

The idea generalizes directly to the case where each processor holds a sorted

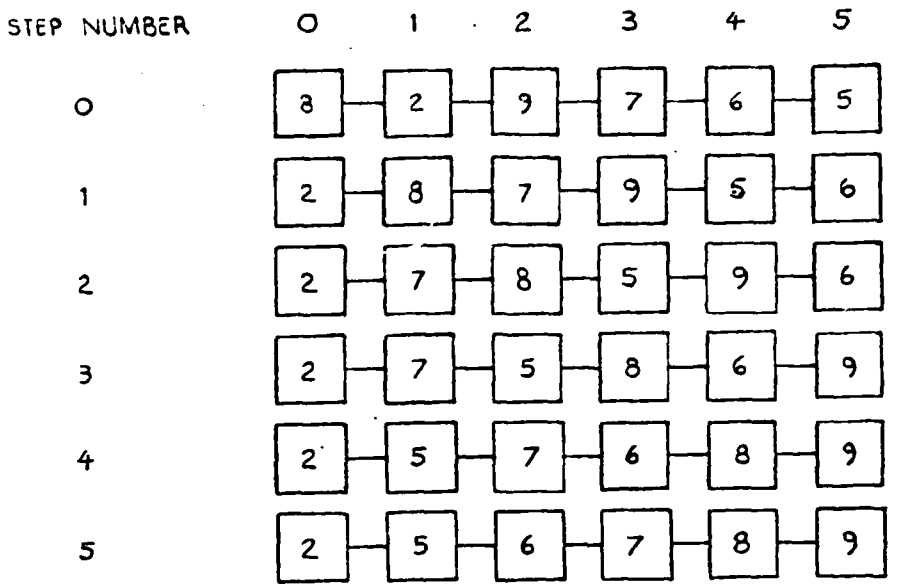


Figure 3-2: The odd-even transposition sort on a one-dimensional linear array.

subsequence of keys rather than a single key [Baudet and Stevenson 78]. For this case, the comparison-exchange operation becomes the merge-splitting operation. Using this generalization, one can sort n keys on k linearly connected processors in $O((n/k)\log(n/k)) + O(k(n/k))$ time, provide that each processor can hold n/k keys (this is possible for SIMD machines). In the above expression, the first term is to the time to sort an (n/k)-subsequence at each processor, and the second term is to the time to perform the odd-even transposition sort on k sorted (n/k)-subsequences. It is readily seen that when n is large relative to k, a speed-up ratio near k is obtained. This near optimal speed-up (with respect to the number of processors used) is due to the fact that when n is large relative to k, the computation done within each processor is large, as compared to interprocessor communication. Thus, the overheads arising from interprocessor communication become relatively insignificant.

Real-Time Finite Impulse Response (FIR) Filtering

One of the most frequently performed computations in signal processing is that of a FIR filter. The computation of a p-tap FIR filter can be viewed as a matrix-vector multiplication where the matrix is a band upper triangular Toeplitz matrix with band width p. Figure 3-3 represents the computation of a 4-tap filter.

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4 & & & 0 \\ & a_1 & a_2 & a_3 & a_4 & & \\ & & a_1 & a_2 & a_3 & a_4 & \\ & & & a_1 & a_2 & a_3 & a_4 \\ & & & & \cdot & & \\ 0 & & & & & \cdot & \\ & & & & & & \cdot \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \cdot \\ \cdot \\ \cdot \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

Figure 3-3: The computation of a 4-tap FIR filter with coefficients a_1, a_2, a_3, and a_4.

In the figure, the sequence $x_1, x_2, x_3 \ldots$ corresponds to a real-time data stream obtained by sampling the signal at times $t, t + \delta, t + 2\delta, \ldots$, and constants a_1, a_2, a_3, and a_4 are the taps of the filter. A p-tap filter can be implemented efficiently by a linear array consisting of p inner product step processors, each capable of performing one multiplication and one add in a unit of time. We illustrate the operation of the linear array by considering the filtering problem in Fig. 3-3. The taps a_i are stored in the array at the beginning of the computation,

one in each processor, and they do not move during the computation (cf. Fig. 3-4). The y_i, which are initially zero, marches to the left, while the x_i are marching to the right. All the moves are synchronized, and the x_i's and y_i's are separated by two time units. It is readily seen that each y_i is able to accumulate all its terms, namely, a_1x_i, a_2x_{i+1}, a_3x_{i+2}, and a_4x_{i+3}, before it leaves the array at the left end processor. Therefore the y_i's are computed in real-time in the sense that they are output in the same rate as the x_i's are input.

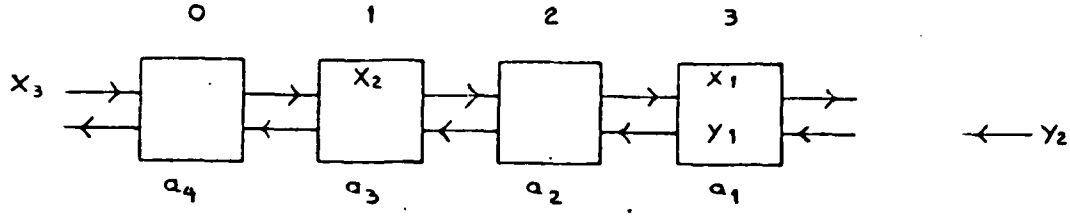


Figure 3-4: The one-dimensional linear array for the computation of the 4-tap filtering in Fig. 3-3.

We now specify the operation of the linear array more precisely. Each processor has three registers, R_a, R_x and R_y, which hold a, x, and y values, respectively. Initially, all R_x and R_y registers contain zeros, and the R_a register at processor i contains the value of a_{4-i}. Each step of the array consists of the following operations, but for odd numbered steps only even numbered processors are activated and for even numbered steps only odd numbered processors are activated.

1. *Shift.*

   - R_x gets the contents of register R_x from the left neighboring processor. (The R_x in processor 0 gets a new component of x.)

   - R_y gets the contents of register R_y from the right neighboring processor. (Processor 0 outputs its R_y contents and the R_y in processor 3 gets zero.)

2. *Multiply and Add.*

   $$R_y \leftarrow R_y + R_a \times R_x.$$

After p units of time final results of the y_i's are pumped out from the left end processor at the rate of one output every two units of time. Fig. 3-5 illustrates four steps of the linear array. Observe that when y_1 is ready to get out from the left end processor at the end of the seventh step, $y_1 = a_1x_1+a_2x_2+ a_3x_3+a_4x_4$, and $y_2 = a_1x_2+a_2x_3$.

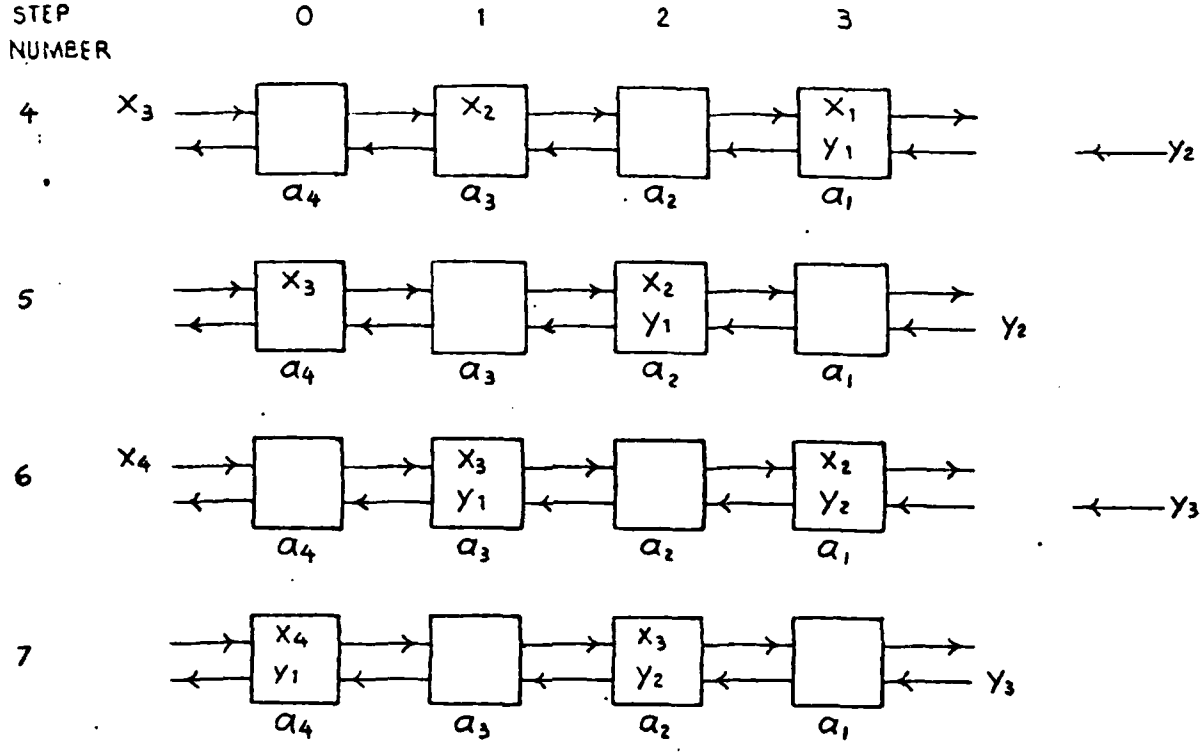


Figure 3-5: Four steps of the linear array in Fig. 3-4.

The FIR result mentioned here is a special case of a result in [Kung and Leiserson 79] concerning linear processor arrays for general matrix-vector multiplications. Similar results hold for the computation of convolutions or discrete Fourier transforms. In general, if A is an nxn matrix of band width w, then a linear array of w processors can multiply A with any n-vector in O(n) time, as compared to O(wn) time needed for a sequential algorithm on a uniprocessor computer.

Priority Queue

A data structure that can process INSERT, DELETE, and EXTRACT_MIN operations is called a priority queue. Priority queues are basic structures used in many programming tasks. If a priority queue is implemented by some balanced tree, for example 2-3 tree, then an operation of the queue will typically take O(log n) time when there are n elements stored in the tree [Aho et al. 75]. This O(log n) delay can be replaced with a constant delay if a linear array of processors is used to implement the priority queue. Here we shall only sketch the

basic idea behind the linear array implementation. A complete description will be reported elsewhere.

To visualize the algorithm, we assume that the linear array in Fig. 3-1 has been physically rotated 90 degrees and that processors are capable of performing comparison-exchange operations on elements in neighboring processors. We try to maintain elements in the array in the sorted order according to their weights. After an element is inserted into the array from the top, it will "sink down" to the proper place by trading positions with elements having smaller weights (so lighter elements will "bubble up"). For deleting an element, we insert an "anti-element" which first sinks down from the top to find the element, and then annihilates it. Elements below can then bubble up into the empty processor. Hence the element with the smallest weight will always appear at the top of the processor array, and is ready to be extracted in constant time. An important observation is that "sinking down" or "bubbling up" operations can be carried out concurrently at various processors throughout the array. For example, the second insertion can start right after the first insertion has passed the top processor. In this way, any sequence of n INSERT, DELETE, or EXTRACT_MIN operations can be done in O(n) time on a linear array of n processors, rather than O(n log n) time as required by a uniprocessor. In particular, by performing n INSERT operations followed by n EXTRACT_MIN operations the array can sort n elements in O(n) time, where the sorting time is completely overlapped with input and output. A similar result on sorting was recently proposed by [Chen et al. 78]. They do not, however, consider the deletion operation.

Recurrence Evaluation (Recursive Filtering)

Many computational tasks such as recursive digit filtering are concerned with evaluations of recurrences. A k-th order recurrence problem is defined as follows: Given $x_0, x_{-1}, \ldots, x_{-k+1}$, compute $x_1, x_2, \ldots$, defined by

$$x_i = R_i(x_{i-1}, \ldots, x_{i-k}) \text{ for } i>0,$$

where the R_i's are given "recurrence functions". For a large class of recurrence functions, a k-th order recurrence problem can be solved in real-time on k linearly connected processors [Kung 79]. That is, a new x_i is output at regular time intervals, at a frequency independent of k. To illustrate the idea, we consider the following linear recurrence:

$$x_i = ax_{i-1} + bx_{i-2} + cx_{i-3} + d,$$

where the a, b, c and d are constants. Clearly feedback links are needed for evaluating such a recurrence on a linear array, since every newly computed term has to be used later for computing other terms. The classical network with feedback loops is depicted in Fig. 3-6.

Each processor (except the right-most one, which has more than one output port) is the inner product step processor similar to the one used before for FIR filtering. The x_i,

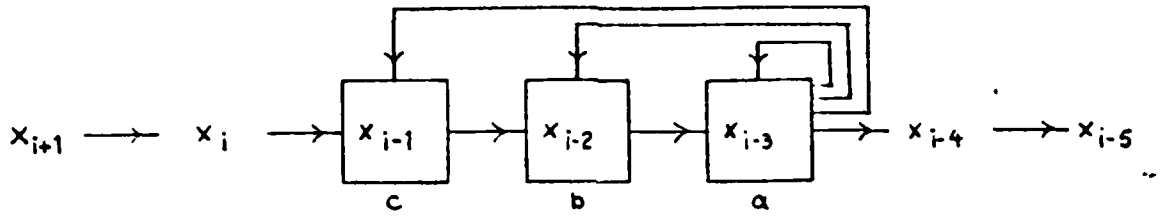


Figure 3-6: A linear array with feedback loops

initialized as d, gets cx_{i-3}, bx_{i-2}, and ax_{i-1} at time 1, 2 and 3, respectively. At time 4, the final value of x_i is output from the right-most processor, and is also fed back to all the processors for use in computing x_{i+1}, x_{i+2} and x_{i+3}. The feedback loops in Fig. 3-6 are undesirable, since they make the network irregular and non-modular. Fortunately, these irregular feedback loops can be replaced with a regular, two-way data flow scheme. Assume that each processor is capable of performing the inner product step and also passes data as depicted in Fig. 3-7 (b). A two-way pipeline algorithm, without irregular feedback loops, for evaluating the linear recurrence is schematized in Fig. 3-7 (a). The additional processor, drawn in dotted lines, passes data only and is essentially a delay. Each x_i enters the right most processor with value zero, accumulates its terms as marching to the left, and feeds back its final value to the array through the left-most processor for use in computing x_{i+1}, x_{i+2} and x_{i+3}. The final values of the x_i's are output from the right-most processor at the rate of one output every two units of time.

This example shows that two-way pipelining is a powerful construct in the sense that it can eliminate undesirable feedback loops as those encountered in Fig. 3-6. Extensions of the two-way pipelining approach to more general recurrence problems are considered in [Kung 79]. Basically the two-way pipelining idea is as follows: By having two data streams travel in opposite directions, a data item in one stream can meet all data items in the other stream and thus their Cartesian product can be formed in parallel in all stages of the pipe. Since Cartesian product-like computations are common in many applications, we expect to find more use of two-way pipelining in the future.

Pipeline Processing of Arithmetic Operations

One of the most successful applications of pipeline processing has been in the execution of arithmetic operations. Pipeline algorithms for floating-point addition, multiplication, division, and square root have been discussed and reviewed in [Chen 75, Ramamoorthy and Li 77]. For these algorithms, the connection among various stages of the "pipe" is by and large

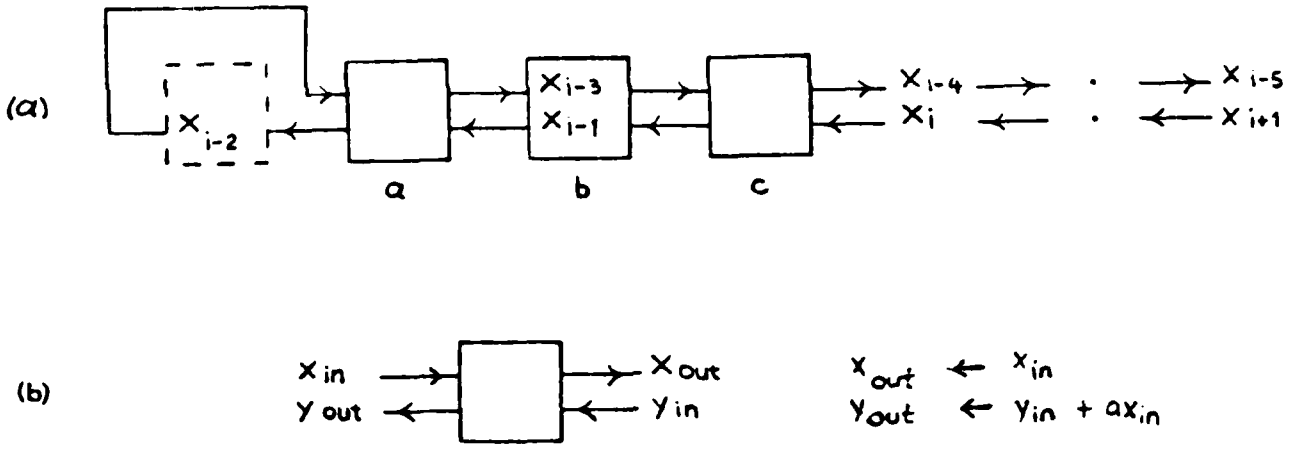


Figure 3-7: (a) A two-way pipeline algorithm without irregular feedback loops, and (b) the inner product step processor.

linear, although additional feedback links may sometimes be present. For example, the Cray Research CRAY-1 uses 6-stage floating-point adders and 7-stage floating-point multipliers, and the CDC STAR-100 uses 4-stage floating-point adders. For a pipeline floating-point adder, the pipe typically consists of stages for performing exponent alignment, fraction shift, fraction addition, and normalization. A pipeline arithmetic unit can be viewed as a systolic machine composed of linearly connected processors that are capable of performing a set of (different) operations.

The pipeline approach is ideal for situations where the same sequence of operations will be invoked very frequently, so that the start-up time to initialize and fill the pipe becomes relatively insignificant. This is the case when the machine is processing long vectors. One of the main concerns in using pipeline machines such as the CRAY-1 and the STAR-100 is the average length of the vectors to be processed (see, for example, [Voigt 77]).

For integer arithmetic, bits in the input operands and carries generated by additions are often pipelined (see, e.g. [Hallin and Flynn 72]). The following pipeline digit-adder using a linear array is described in [Chen 75]. Suppose that we want to add two integer vectors (U_1, U_2, . . .) and (V_1, V_2, . . .), and that $U_i = u_{i1}u_{i2}\ldots u_{ik}$ and $V_i = v_{i1}v_{i2}\ldots v_{ik}$ in their binary representations. We illustrate how the adder works for k = 3 in Fig. 3-8. The u_{ij} and v_{ij} march toward the processors synchronously as shown.

At each cycle, each processor sums the three numbers arriving from the three input lines and then outputs the sum and the carry at the output lines. It is easy to check that with the

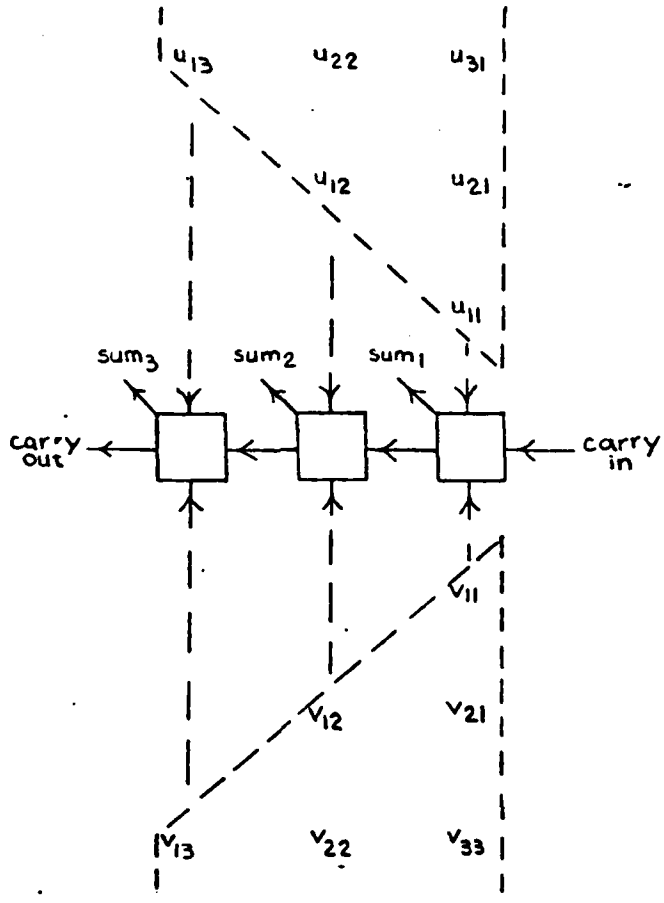


Figure 3-8: A pipeline integer adder

configuration shown, when the pair (u_{ij}, v_{ij}) reaches a processor, the carry needed to produce the correct j-th digit in the result of U_i+V_i, will also reach the same processor. As a result, the pipelined adder can compute a sum $U_i + V_i$ every cycle in the steady state.

3.3 Algorithms Using Two-Dimensional Arrays

We restrict ourselves to two-dimensional communication geometries which are simple and regular. Consider the following problem: how can processors be distributed in a two-dimensional area so that they can be mesh-connected in a simple and regular way, in the sense that the connections are all symmetric and of the same length? It turns out that there are only three solutions to the problem. This problem is related to that of finding regular figures which can close pack to completely cover a two-dimensional area. The only three

regular figures which possess this property are the square, the hexagon and the equilateral triangle (see Fig. 3-9). In the following, we consider algorithms using hexagonal and square arrays. Interesting algorithms using equilateral triangular arrays are yet to be discovered.

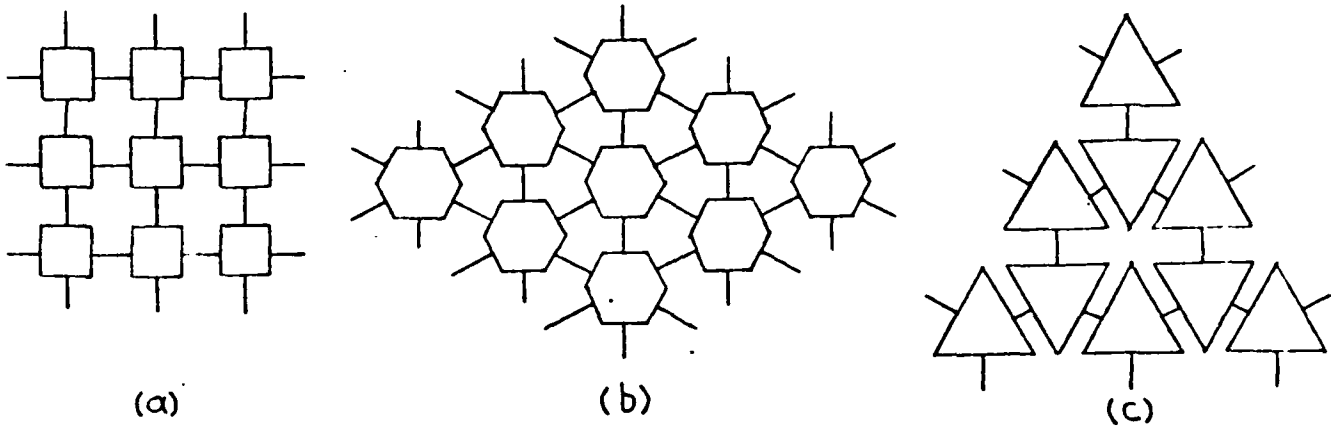


Figure 3-9: The three types of regular arrays:
(a) square array, (b) hexagonal array, (c) triangular array.

3.3.1 Algorithms Using Two-Dimensional Hexagonal Arrays

We demonstrate that two matrix algorithms, matrix multiplication and LU-decomposition, can be done naturally on hexagonal arrays. The basic processor used by these two algorithms is the inner product step processor (Fig. 3-10), which is similar to the ones used in Section 3.2 for FIR filtering and recurrence evaluations. The processor has three registers R_A, R_B, and R_C, and has six external connections, three for input and three for output. In each unit time interval, the processor shifts the data on its input lines denoted by A, B and C into R_A, R_B and R_C, respectively, computes $R_C \leftarrow R_C + R_A \times R_B$, and makes the input values for R_A and R_B together with the new value of R_C available as outputs on the output lines denoted by A, B and C, respectively. All outputs are latched and the logic is clocked so that when one processor is connected to another, the changing output of one during a unit time interval will not interfere with the input to another during this time interval. This is not the only processing element we shall make use of, but it will be the work horse. A special processor for computing reciprocals will be specified later when it is used. For details about these two algorithms and other related results, see [Kung and Leiserson 78, Kung and Leiserson 79]. The hexagonal array connection is also natural for computing the transitive closure of a Boolean matrix. In this case, the inner product step processor computes $R_C \leftarrow R_C \vee R_A \wedge R_B$.

Other examples of computations using hexagonal arrays include QR-factorization [Brent and Kung 79a], relational database operations [Kung and Lehman 79a], and the tally circuit [Mead

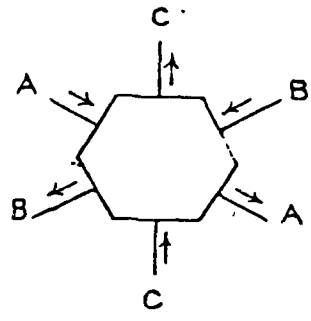


Figure 3-10: The inner product step processor.

and Conway 79].

Matrix Multiplication

$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & \ddots & \ddots & \\ 0 & & \ddots & \ddots & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} & \\ & b_{32} & b_{33} & b_{34} & b_{35} \\ & & b_{42} & \ddots & \ddots \\ 0 & & & \ddots & \ddots \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & 0 \\ c_{21} & c_{22} & c_{23} & c_{24} & \\ c_{31} & c_{32} & c_{33} & c_{34} & \\ c_{41} & c_{42} & \ddots & \ddots & \\ 0 & & \ddots & \ddots & \end{bmatrix}$$

A B C

Figure 3-11: Band matrix multiplication.

It is easy to see that the matrix product $C = (c_{ij})$ of $A = (a_{ij})$ and $B = (b_{ij})$ can be computed by the following recurrences.

$$c_{ij}^{(1)} = 0,$$

$$c_{ij}^{(k+1)} = c_{ij}^{(k)} + a_{ik}b_{kj},\ k=1, 2, \ldots, n,$$

$$c_{ij} = c_{ij}^{(n+1)}.$$

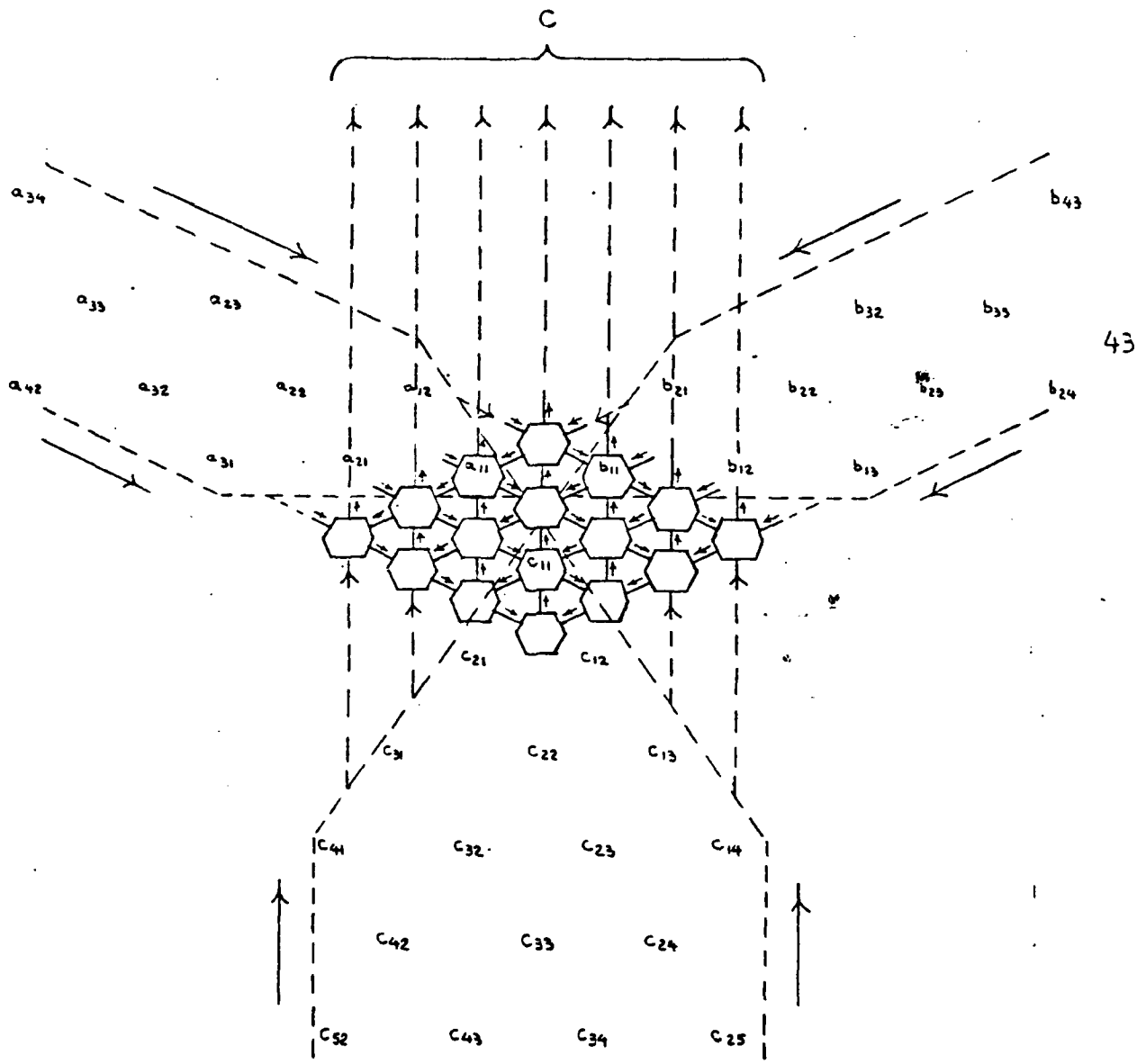
Let A and B be nxn band matrices of band width w_1 and w_2, respectively. We show how the recurrences above can be evaluated by pipelining the a_{ij}, b_{ij} and c_{ij} through an array of w_1w_2 hex-connected inner product step processors. We illustrate the algorithm by considering the band matrix multiplication problem in Fig. 3-11. The diamond shaped hexagonal array for this case is shown in Fig. 3-12, where arrows indicate the directions of data flow.

The elements in the bands of A, B and C march through the network in three directions synchronously. Each c_{ij} is initialized to zero as it enters the network through the bottom boundaries. (For the general problem of computing C = AB + D where D = (d_{ij}) is any given matrix, c_{ij} should be initialized as d_{ij}.) One can easily see that with the inner product step processors depicted in Fig. 3-10, each c_{ij} is able to accumulate all its terms before it leaves the network through the upper boundaries. If A and B are nxn band matrices of band width w_1 and w_2, respectively, then an array of w_1w_2 hex-connected processors can pipeline the matrix multiplication AxB in $3n+\min(w_1, w_2)$ units of time. If A and B are nxn dense matrices then $3n^2-3n+1$ hex-connected processors can compute AxB in 5(n-1) units of time. We mention an important application of this result. It is well-known that an n^2-point discrete Fourier transform (DFT) can be computed by first performing n independent n-point DFT's and then using the results to perform another set of n independent n-point DFT's. The computation of any of these two sets of n independent n-point DFT's is simply a matrix multiplication AxB, where the (i,j) entry of matrix A is $\omega^{(i-1)(j-1)}$ and ω is a primitive nth root of unity. Hence, using $O(n^2)$ hex-connected processors, an n^2-point DFT can be computed in O(n) time.

The LU-Decomposition of a Matrix

The problem of factoring a matrix A into lower and upper triangular matrices L and U is called LU-decomposition. Figure 3-13 illustrates the LU-decomposition of a band matrix with p = 4 and q = 4. Once the L and U factors are known, it is relatively easy to invert A or to solve the linear system Ax = b.

We assume that matrix A has the property that its LU-decomposition can be done by Gaussian elimination without pivoting. (This is true, for example, when A is a symmetric positive-definite, or an irreducible, diagonally dominant matrix.) The triangular matrices L = (l_{ij}) and U = (u_{ij}) are evaluated according to the following recurrences.



43

Figure 3-12: The hexagonal array for the matrix multiplication problem in Fig. 3-11.

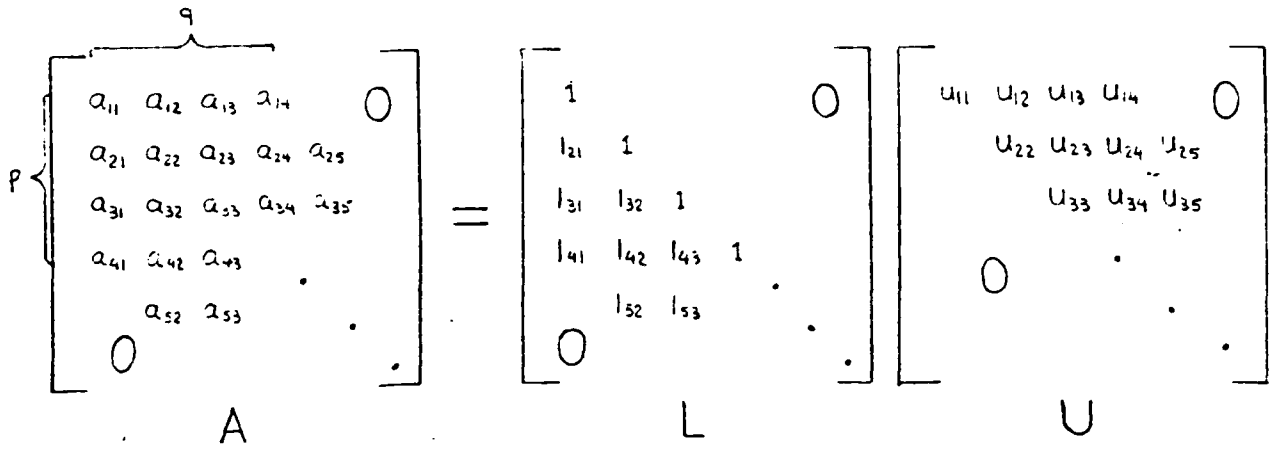


Figure 3-13: The LU-decomposition of a band matrix.

$$a_{ij}^{(1)} = a_{ij},$$

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} + l_{ik}(-u_{kj}),$$

$$l_{ik} = \begin{cases} 0 & \text{if } i < k, \\ 1 & \text{if } i = k, \\ a_{ik}^{(k)} u_{kk}^{-1} & \text{if } i > k, \end{cases}$$

$$u_{kj} = \begin{cases} 0 & \text{if } k > j, \\ a_{kj}^{(k)} & \text{if } k \le j. \end{cases}$$

It turns out that the evaluation of these recurrences can be pipelined on a hexagonal array. A global view of this pipelined computation is shown in Fig. 3-14 for the LU-decomposition problem in Fig. 3-13. The array in Fig. 3-14 is constructed as follows. The processors below the upper boundaries are the standard inner product step processors and are hex-connected exactly the same as the matrix multiplication network presented above. The processor at the top, denoted by a circle, is a special processor. It computes the reciprocal of its input and pumps the result southwest, and also pumps the same input northward unchanged. The other processors on the upper boundaries are again inner product step processors, but their

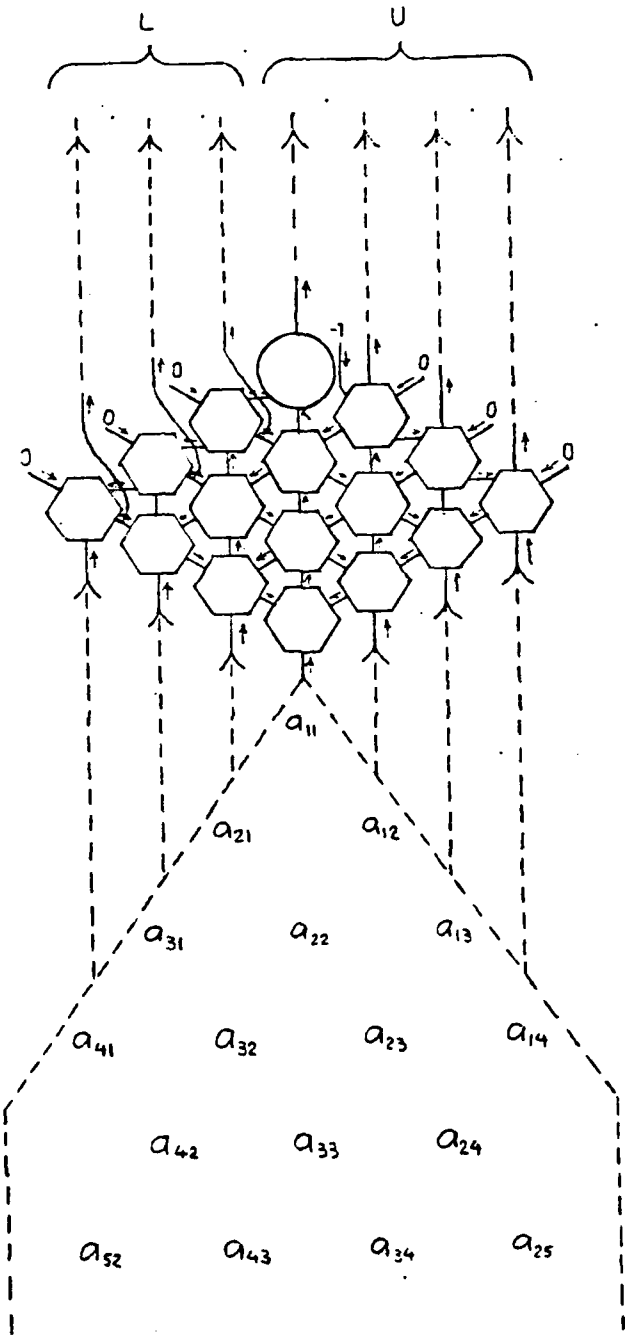


Figure 3-14: The hexagonal array for pipelining the LU-decomposition of the band matrix in Fig. 3-13.

orientation is changed: the ones on the upper left boundary are rotated 120 degrees clockwise; the ones on the upper right boundary are rotated 120 degrees counterclockwise. The flow of data in the array is indicated by arrows in the figure.

If A is an nxn band matrix with band width w = p+q-1, an array having no more than pq hex-connected processors can compute the LU-decomposition of A in 3n+min(p,q) units of time. If A is an nxn dense matrix, then an nxn hexagonal array can compute the L and U matrices in 4n-2 units of time which includes I/O time. The remarkable fact that the matrix multiplication network forms a major part of the LU-decomposition network is due to the similarity of the defining recurrences.

Transitive Closure

Given a Boolean matrix $A=(a_{ij})$, the transitive closure of A can be computed from the recurrence:

$$a_{ij}^{(1)} = a_{ij},$$
$$a_{ij}^{(k+1)} = a_{ij}^{(k)} \vee (a_{ik}^{(k)} \wedge a_{kj}^{(k)}),$$

(see, e.g., [Aho et al. 75]). We observe that this recurrence is analogous to the recurrence for matrix multiplication or LU-decomposition, as far as structures for subscripts and superscripts are concerned. This suggests that we use hexagonal arrays to solve the transitive closure problem, too. Indeed, an efficient transitive closure algorithm using the hexagonal array has recently been discovered. The algorithm differs from the matrix multiplication and LU-decomposition algorithm in that it computes the solution in two passes rather than one pass. A full description of the algorithm will appear in the revised version of [Guibas et al. 79].

3.3.2 Algorithms Using Two-dimensional Square Arrays

The square array is perhaps one of the first communication geometries studied by researchers who were interested in parallel processing. Work in cellular automata, which is concerned with computations distributed in a two-dimensional orthogonally connected array, was initiated by von Neumann in the early fifties [Von Neumann 66]. Theorists in cellular automata have been traditionally interested in the "power" of a cellular automaton system using, say, a particular number of states at each cell. More recently, because of the advent of LSI technology, there has been an increasing interest in designing algorithms for cellular arrays. Cellular algorithms for pattern recognition have been proposed in [Smith 71, Kosaraju 75, Foster and Kung 79], for graph problems in [Levitt and Kautz 72], for

switching in [Kautz et al. 68], for sorting in [Thompson and Kung 77], and for dynamic programming in [Guibas et al. 79]. The algorithms for dynamic programming in [Guibas et al. 79] are quite special in that they involve data being transmitted at two different speeds, which give the effect of "time reverse" for the order of certain results. The pattern matching chip described in [Foster and Kung 79] has recently been designed and fabricated.

In parallel to the developments of cellular algorithms for solving combinatorial problems, there have been major activities in using the array structure for solving large numerical problems. Many of these activities are motivated or influenced by the ILLIAC IV computer, which has an 8x8 processor array (see [Kuck 68]). Relaxation methods for solving partial differential equations match the square array structure naturally. Typically, the variable u_{ij} representing the solution at mesh point (i, j) is updated by a difference equation of the form:

$$u_{ij} = F(u_{i+1, j}, u_{i-1, j}, u_{i, j+1}, u_{i, j-1}).$$

Hence, if u_{ij} is stored at processor (i, j) of the processor array, then each update (or iteration) involves communications only among neighboring processors. The central control provided by SIMD machines such as the ILLIAC IV is useful for broadcasting relaxation and termination parameters, which are often needed in these relaxation methods. Relaxation algorithms on two-dimensional grids are also used in image processing, for which mesh points correspond to pixels [Peleg and Rosenfeld 78].

3.4 Algorithms Using Tree Structures

The tree structure, shown in Fig. 3-15 (a), has the nice property that it supports logarithmic-time broadcast, search, and fan-in. Fig. 3-15 (b) shows an interesting "H" shaped layout of a binary tree, which is convenient for placement on a chip.

Unlike the array structures considered earlier, the connections in the tree structure are not uniform. The distance between two connecting processors increases as they move up to the root. For chip implementation, the time that it takes a signal to propagate along a wire can nevertheless be made independent of the length of the wire, by fitting larger drivers to longer wires. Thus, by using appropriate drivers the logarithmic property of the tree structure can still be maintained. It is demonstrated in [Mead and Rem 79] that in spite of the fact that large drivers take large areas, with the layout in Fig. 3-15 (b) it is possible to implement a tree using a total chip area essentially proportional to the number of processors in the tree. Moreover, in this implementation drive currents ramp up from the leaves to the root, and consequently, off-chip communication can be conducted at the root without serious delay. In the following, we shall assume that the time to send a data item across any link in the tree is constant, and that the root of the tree is the I/O node for outside world

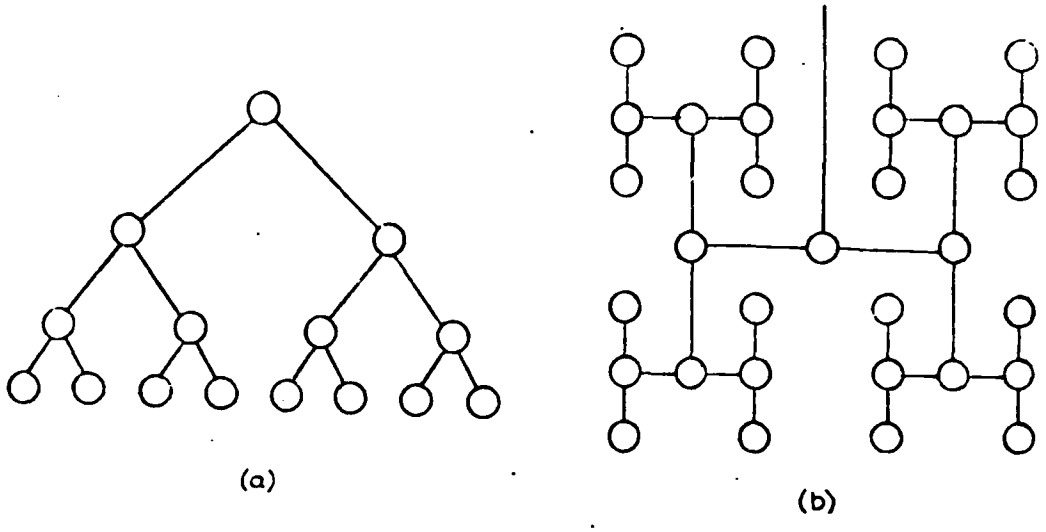


Figure 3-15: (a) A binary tree structure, and (b) embedding a binary tree in a two-dimensional grid.

communication.

The logarithmic-time property for broadcasting, searching, and fan-in is the main advantage provided by the tree structure that is not shared by any array structure. The tree structure, however, has the following possible drawback. Processors at high levels of the tree may become bottlenecks if the majority of communications are not confined to processors at low levels. We are interested in algorithms that can take advantage of the power provided by the tree structure while avoiding its drawback.

Search Algorithms

The tree structure is ideal for searching. Assume, for example, that information stored at the leaves of a tree forms the data base. Then we can answer questions of the following kinds rapidly: "What is the nearest neighbor of a given element?", "What is the rank of a given element?", "Does a given element belong to a certain subset of the data base?" The paradigm to process these queries consists of three phases: (i) the given element is broadcast from the root to leaves, (ii) the element is compared to some relevant data at every leaf simultaneously, and (iii) the comparison results from all the leaves are combined into a single answer at the root, through some fan-in process. It should be clear that using the paradigm and assuming appropriate capabilities of the processors, queries like the ones above can all be answered in logarithmic time. Furthermore, we note that when there are

many queries, it is possible to pipeline them on the tree. See [Bentley and Kung 79] for discussions of using the tree-structured machine for many searching problems.

A similar idea has been pointed out in [Browning 79]. Algorithms which first generate a large number of solution candidates and then select from among them the true solutions can be supported by the tree structure. NP-complete problems [Karp 72] such as the clique problem and the color cost problem are solvable by such algorithms. One notes immediately that with this approach an exponential number of processors will be needed to solve an NP-complete problem in polynomial time. However, with the emergence of very large scale integration (VLSI) technology this brute force approach may gain importance. Here we merely wish to point out that the tree structure matches the structure of some algorithms that solve NP-complete problems.

Systolic Search Tree

As one is thinking about applications using trees, data structures such as search trees (see, e.g., [Aho et al. 75, Knuth 73]) will certainly come to mind. The problem is how to embed a balanced search tree in a network of processors connected by a tree so that the O(log n) performance for the INSERT, DELETE, and FIND operations can be maintained. The problem is nontrivial because most balancing schemes require moving pointers, but the movement of pointers is impossible in a physical tree where pointers are fixed wires. To get the effect of balancing in the physical tree, data rather than pointers must be moved. Common balanced tree schemes such as AVL trees and 2-3 trees do not map well onto the tree network because data movements involved in balancing are highly non-local. A new organization of a hardware search tree, called a systolic search tree, was recently proposed by [Leiserson 79], on which the data movements for balancing are always local so that the desired O(log n) performance can be achieved. In Leiserson's paper an application of using the systolic search tree as a common storage for a collection of disjoint priority queues is discussed.

Evaluation of Arithmetic Expressions and Recurrences

Another application of the tree structure is its use for evaluating arithmetic expressions. Any expression of n variables can be evaluated by a tree of at most $4\lceil \log_2 n \rceil$ levels [Brent 74], but the time to input the n variables to the tree from the root is still O(n). This input time can often be overlapped with the computation time in the case of recurrences evaluation (see [Kung 79]).

3.5 Algorithms Using Shuffle-Exchange Networks

Consider a network having $n=2^m$ nodes, where m is an integer. Assume that nodes are named as $0, 1, \ldots, 2^m-1$. Let $i_m i_{m-1} \ldots i_1$ denote the binary representation of any integer i, $0 \le i \le 2^{m-1}-1$. The shuffle function is defined by

$$S(i_m i_{m-1} \ldots i_1) = i_{m-1} i_{m-2} \ldots i_1 i_m,$$

and the exchange function is defined by

$$E(i_m i_{m-1} \ldots i_1) = i_m i_{m-1} \ldots i_2 \bar{i}_1.$$

The network is called a shuffle-exchange network if node i is connected to node $S(i)$ for all i, and to node $E(i)$ for all even i. It is often convenient to view each pair of nodes connected by the exchange function as a 2x2 processor which has two input ports and two output ports. Fig. 3-16 illustrates the shuffle function for the case when $n=2^m=8$.

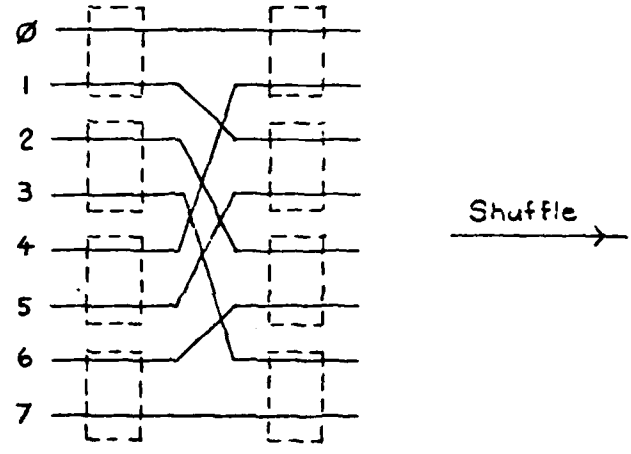


Figure 3-16: The shuffle function and 2x2 processors for the case when n=8.

Observe that for $i=1, \ldots, m$, by executing the shuffle function i times, data originally at two nodes whose names differ by 2^{m-i} can be brought to the same 2x2 processor. This type of communication happens to be natural to a number of algorithms. It was shown by [Batcher 68] that the bitonic sort of n elements can be carried out in $O(\log^2 n)$ steps on the shuffle-exchange network when the 2x2 processors are capable of performing comparison-exchange operations. It was shown by [Pease 68] that the n-point fast Fourier transform (FFT) can be done in $O(\log n)$ steps on the network when the 2x2 processors are capable of doing addition and multiplication operations. Other applications including matrix transposition and linear recurrence evaluation are given in [Stone 71, Stone 75]. The two

articles by Stone give clear expositions for all these algorithms and have good discussions on the basic idea behind them. Here we illustrate the use of the network for performing the 8-point FFT. The computation has three stages. Stage i, i=1, 2, 3 involve combining data from two nodes whose names differ by 2^{3-i}. This is indicated by the graph in Fig. 3-17 (a). A topologically equivalent graph is shown in Fig. 3-17 (b). The latter graph demonstrates the fact that the computation at each stage can be done entirely inside the 2x2 processors, provided that results from the previous stage have been "shuffled". Note that the same shuffle network can be used for shuffling inputs for all the stages if so desired.

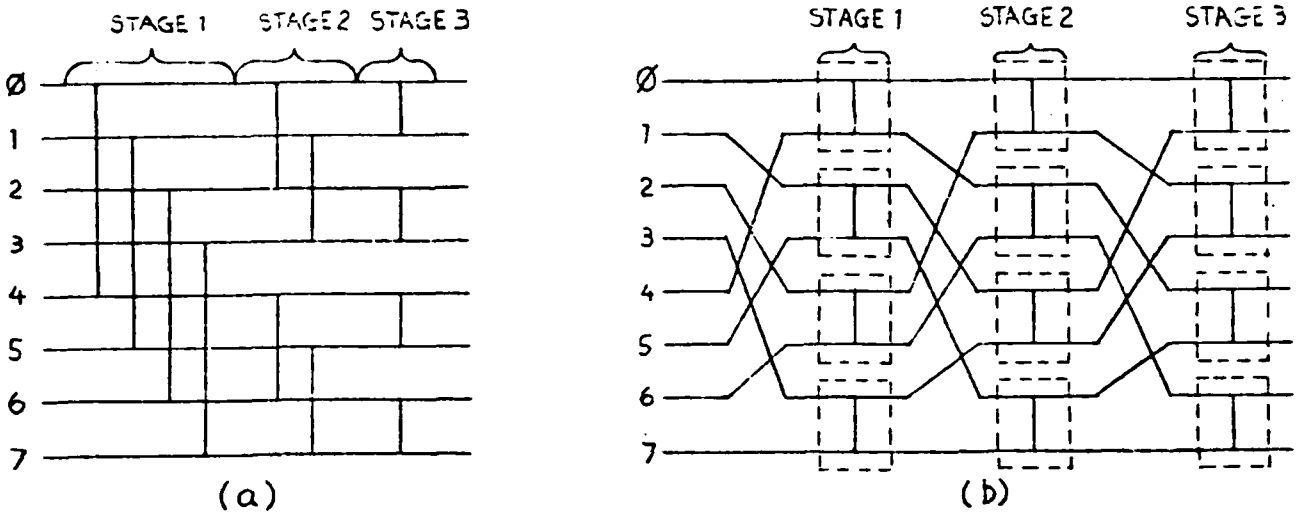


Figure 3-17: (a) The communication structure of the 8-point FFT, and (b) its realization by the shuffle-exchange network.

Many powerful rearrangeable permutation networks, such as those in [Benes 65] which are capable of performing all possible permutations in O(log n) delays, can be viewed as multi-stage shuffle-exchange networks (see, e.g., [Kuck 78]). The shuffle-exchange network, perhaps due to its great power in permutation, suffers from the drawback that it has a very low degree of regularity and modularity. Indeed, it was recently shown by [Thompson 79a] that the network is not planar and cannot be embedded in silicon using area linearly proportional to the number of nodes in the network.

3.6 Remarks for Section 3

For a fixed problem, it is often possible to design algorithms using different communication topologies. A good example of this is the sorting problem. A performance hierarchy for sorting n elements on n processors connected by various networks is given in Fig. 3-18 [Knuth 73, Thompson and Kung 77, Batcher 68].

NETWORK	SORTING TIME
1-dim array	$\theta(n)$
2-dim array	$\theta(n^{1/2})$
k-dim array	$\theta(n^{1/k})$
Shuffle-Exchange	$O(\log^2 n)$

Figure 3-18: Sorting times on various networks

Each of these algorithms can be useful under appropriate circumstances. For a discussion on the related problem of mapping a given algorithm (rather than a given problem) on different networks, see [Kung and Stevenson 77].

Sorting can also be done on the tree structure in O(n) time in a straightforward way. But the same performance is achievable by the simpler one-dimensional linear array using the priority queue approach (cf. Section 3.1.). For this reason, we did not include tree sort as one of the algorithms for the tree network in Section 3.4. The general guideline we have been using in this section for choosing algorithms under a given communication structure is as follows: An algorithm is included only if it uses the structure effectively, in the sense that the same performance does not seem to be possible on a simpler structure. One should note, however, that sometimes it may be worthwhile to consider solving a problem on some network which is not inherently best suited for the problem. For instance, at an installation a fixed network may have to be used for solving a set of rather incompatible problems.

Up to this point, we have been considering almost exclusively the case when there are enough processors for the problem one wants to solve. The only exception is that for the odd-transposition sort we discussed how to sort n elements by k processors where $k<n$, and concluded that a near-optimal speed up ratio can be achieved if $k<<n$. In general, there are three approaches one can take for solving a large problem on a small network.

i. Use algorithms with large module granularity. Each processor handles a large group of elements rather than a few elements. For the odd-even transposition

sort mentioned above, a subsequence consisting of n/k elements is stored in each processor. For matrix problems, a row, a column, or a submatrix may be stored in a processor. This approach is suitable for SIMD machines where processors can have relatively large local memories. In this case, one must carefully design the global data structure, which is now distributed over the local memories, so as to ensure that needed memory accesses can be performed in parallel without conflicts [Lawrie 75, Kuck 78].

ii. Decompose the problem. The idea is that after decomposition each subproblem will be small enough so that it can be solved on the given small network of processors. A matrix multiplication involving large matrices, for example, can be done on a small network by performing a sequence of matrix multiplications involving submatrices.

iii. Decompose an algorithm that originally requires a large network. Simultaneous operations invoked in one step of the original algorithm are now carried out in a number of steps by the small network. With this approach, the LU-decomposition algorithm for an nxn matrix in section 3.3.1 can be performed on a kxk hexagonal array in $O(n^3/k^2)$ time, when n is large and k is fixed.

Using one of the three approaches, one should be able, in principle, to design algorithms for small networks to solve large problems.

4. Algorithms for Asynchronous Multiprocessors

4.1 Introduction

In this section we consider parallel algorithms for an asynchronous MIMD multiprocessor like C.mmp or Cm*, which is composed of a number of independent processors sharing the primary memory by means of a switch or connecting network. Such an algorithm will be viewed as a collection of cooperating processes that may execute simultaneously in solving a given problem. It is important to distinguish between the notion of *process*, which corresponds to the execution of a program, and the notion of *processor*, which is a functional unit by which a process can be carried out. At the decision of the operating system, the same process may be executed by any processor at a given time.

In the design and analysis of parallel algorithms for asynchronous multiprocessors, one should assume that the time required to execute the steps of a process is unpredictable [Kung 76]. Based on measurements obtained from C.mmp, six major sources for causing fluctuations in execution times have been identified [Oleinick 78]. The six sources include variations in computation time due to different instances of inputs, memory contention, operating system's scheduling policies, variations in the individual processor speeds, etc. This asynchronous behavior leads to serious issues regarding the correctness and efficiency of an algorithm. The correctness issue arises because during the execution of an algorithm operations from different processes may interleave in an unpredictable manner. The efficiency issue arises because any synchronization introduced for correctness reasons takes extra time and also reduces concurrency. In the following, we shall examine various techniques for dealing with the correctness and efficiency issues that are encountered in the use of asynchronous multiprocessors.

Asynchronous multiprocessors can support truly concurrent database systems, where simultaneous access to a database by more than one process is possible. Recent research results concerning the integrity of multi-user database systems are directly applicable to concurrent database systems. Some of these results will be examined in Section 4.2. A concurrent database system can be viewed as an asynchronous algorithm consisting of processes that execute so-called transactions. In designing a general database system, one usually has little control over the set of transactions that will be allowed to run in the system. However, in designing an algorithm to solve a fixed problem, one does have control over the tasks to be included in the algorithm. As a result, it is often possible to design parallel algorithms without costly synchronizations for solving specific problems. We shall consider several of these highly efficient algorithms in Section 4.3. Finally, in Section 4.4, we shall discuss some of the guidelines for designing efficient algorithms for asynchronous

multiprocessors.

4.2 Concurrent Database Systems

In a concurrent database system, a number of on-line transactions are allowed to run concurrently on a shared database. One of the important issues arising from the concurrent execution of transactions is the consistency problem. A database is said to be consistent if all integrity constraints defined for the data are met. A transaction is said to be correct if starting from a consistent state the execution of the transaction will terminate and preserve consistency. A concurrent execution of several correct transactions may, however, transform a consistent database into an inconsistent one! We illustrate this fact with a simple example. Suppose that the integrity constraint is $x > 0$. Then the transaction, *if* $x > 1$ *then* $x \leftarrow x - 1$, is correct, but the concurrent execution of two such transactions may transform a consistent state, $x = 2$, into an inconsistent state, $x = 0$. The mechanism in a concurrent database system that safeguards database consistency is usually called a "concurrency control". (In Section 2, we have used the same term with a more general meaning.)

There have been two major approaches in contending with the consistency problem. The first approach, discussed in Section 4.2.1 below, is the "serialization approach", which requires no knowledge of the integrity constraints, but does require syntactic information about the transactions. The second approach, considered in Section 4.2.2, will use specific knowledge of the integrity constraints to construct correct and hopefully more efficient concurrent database systems. In [Kung and Papadimitriou 79] maximum degrees of concurrency are proved to depend upon the types of knowledge that are available.

Besides the consistency issue, there are a number of other important issues concerning concurrent database systems. Among them is the recovery problem. Solution of the recovery problem often closely related to solutions to the consistency problem. The recovery problem will not be explicitly treated in this paper. The reader is referred to [Gray 78] for a good discussion of recovery.

4.2.1 The Serialization Approach

Throughout our discussion, transactions are assumed to be correct in the sense that they preserve database consistency when executed alone. Serial execution of a set of transactions is one-transaction-at-a-time execution. It preserves consistency, since the execution of each transaction does so (see Fig. 4-1).

The serialization approach makes sure that a concurrent execution has the same overall effect as some serial execution and therefore preserves consistency. This approach is very

Figure 4-1: A serial execution of correct transactions, T_1, T_2 . . ., which preserves consistency.

general in the sense that it applies to any concurrent database system and requires no information on the semantics of the transactions and integrity constraints. In fact, it has been shown in [Kung and Papadimitriou 79] that serialization is the weakest criterion for preserving consistency if only syntactic information can be used.

4.2.1.1 The Two-Phase Transaction Method

In [Eswaran et al. 76] a serialization method is proposed in which each transaction employs a locking protocol to insure that it "sees" only a consistent state of the database. Here we briefly describe their scheme. It is assumed that a transaction must have a share lock or exclusive lock on any entity it is reading, and an exclusive lock on any entity it is writing. Fig. 4-2 shows the compatibility among the lock modes.

	SHARE	EXCLUSIVE
SHARE	YES	NO
EXCLUSIVE	NO	NO

Figure 4-2: Compatibilities among lock modes.

A transaction is a <u>two-phase</u> transaction if it does not request new locks after releasing a lock. Hence a two-phase transaction consists of a growing phase during which it requests locks, and a shrinking phase during which it releases locks. A <u>schedule</u> of a set of concurrent transactions is a history of the order in which statements in the transactions are executed. A schedule can be totally ordered or partially ordered. The latter case corresponds to the multiprocessor environment where a set of statements from different transactions can be executed simultaneously by a number of processors. A <u>serial schedule</u> is a schedule corresponding to a serial execution of the transactions. A schedule is <u>legal</u> if it does not schedule a lock action on an entity for one transaction when that entity is already locked by some other transaction in a conflicting mode. In Fig. 4-3 we illustrate a possible legal schedule of two-phase transactions T_1 and T_2.

The numbering in the left hand side specifies the execution order of the schedule (so the

	T_1	T_2
1.	exclusive lock x	
2.	exclusive lock y	
3.	read x	
4.	read y	
5.	write x	
6.	unlock x	
7.		(B) exclusive lock x
8.		(B) write x
9.	(A) write y	
10.	(A) unlock y	
11.		share lock y
12.		unlock x
13.		read y
14.		unlock y

Figure 4-3: A legal schedule of two-phase transactions T_1 and T_2.

schedule is totally ordered). Note that actions in (A) are independent from actions in (B) in the sense that (A) and (B) involve disjoint variables. Thus, the input and output of any action in (A) or (B) is unchanged if actions in (A) precede actions in (B) instead. This implies that the effect of the schedule is the same as that of the serial schedule that executes T_1 first and then T_2. Theorem 4-1 below asserts that the same phenomenon holds for any legal schedule of two-phase transactions. To understand the theorem, we need to introduce some additional terminology. The dependency graph of a schedule is a directed graph whose nodes are transaction names and whose arcs, which are labeled, indicate how transactions depend on each other. More precisely, an arc from T_i to T_j exists if and only if during the execution T_j reads an entity T_i has written or T_j writes an entity T_i has read or written, and the label of the arc in this case is the name of the entity. The dependency graph of a schedule completely determines the state of the database each transaction "sees" when transactions are executed according to the schedule. We say two schedules are equivalent if they have the same dependency graph. We state the main theorem regarding the two-phase transaction method:

> Theorem 4-1: Any legal schedule of two-phase transactions is equivalent to a serial schedule.

The theorem implies that at the termination of any concurrent execution of two-phase

transactions the consistency of the database is maintained. A concurrent execution of two-phase transactions may lead to a deadlock however. In this case, after the deadlock is detected, any transaction on the deadlock cycle can be backed up. Because all the transactions are two-phase locked, it is guaranteed (why?) that backing up a transaction for breaking a deadlock will neither cause other transactions to lose updates, nor require backing up other transactions.

Much insight into locking can be gained by a simple geometric method [Kung and Papadimitriou 79]. Consider the concurrent execution of two transactions T_1 and T_2. Any state of progress towards the completion of T_1 and T_2 can be viewed as a point in the two-dimensional "progress space", as shown in Fig. 4-4.

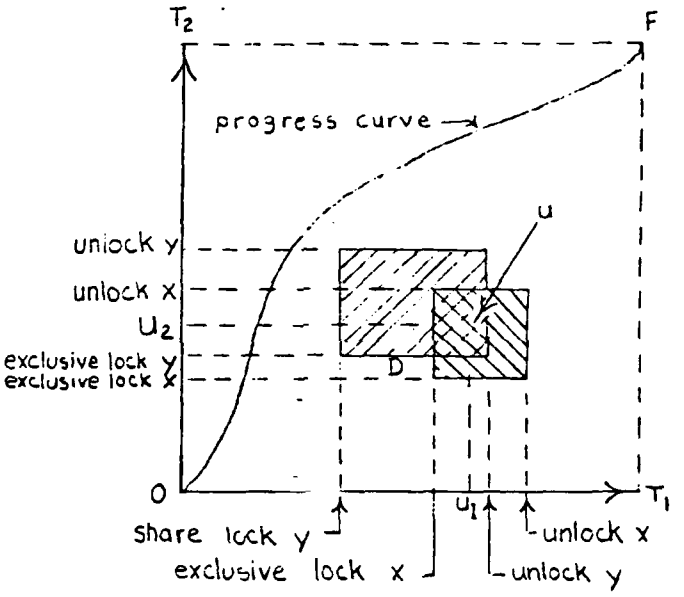


Figure 4-4: The "progress space" for transactions T_1 and T_2.

A schedule of T_1 and T_2 corresponds to a nondecreasing curve, called a progress curve, from the origin to point F. The progress curves lying on the two boundaries, OT_2F and OT_1F, represent the two serial schedules. Locking has the effect of imposing restrictions in the form of forbidden rectangular regions (see the two blocks in Fig. 4-4). It is easy to see that a schedule is legal if and only if its progress curve avoids all blocks. Region D in the figure is a deadlock region, in the sense that any progress curve trapped in the region will not be able to reach F. The important observation is that two schedules are equivalent if and only if their progress curves are not separated by any block. Consequently, if all the blocks are

connected as in Fig. 4-4, then any legal schedule (which avoid blocks) must be equivalent to some serial schedule. The idea of two-phase transactions is now extremely easy to explain. It simply keeps all blocks connected by letting them have a point u in common. The coordinates u_1, u_2 of u are the phase-shift points, at which all locks have been granted, and none have been released.

4.2.1.2 Validation Methods - An Optimistic Approach

Validation methods represent another general approach for achieving serialization [Kung and Robinson 79]. The methods rely on transaction backup rather than locking as a control mechanism. The methods are "optimistic" in the sense that they "hope" that conflicts between transactions will not occur and thus transaction backup will not be necessary. The idea behind this optimistic approach is quite simple, and may be summarized as follows:

- Since reading a value or a pointer from a node can never cause a loss of integrity, reads are completely unrestricted (however, returning a result from a query is considered to be equivalent to a write, and so is subject to validation as discussed below).

- Writes are severely restricted. It is required that any transaction consist of two or three phases: a read phase, a validation phase, and a possible write phase (see Fig. 4-5). During the read phase, all writes take place on local copies of the nodes to be modified. Then, if it can be established during the validation phase that the changes the transaction made will not cause a loss of integrity, the local copies are made global in the write phase. In the case of a query, it must be determined that the result the query would return is actually correct. The step in which it is determined that the transaction will not cause a loss of integrity (or that it will return the correct result) is called *validation*.

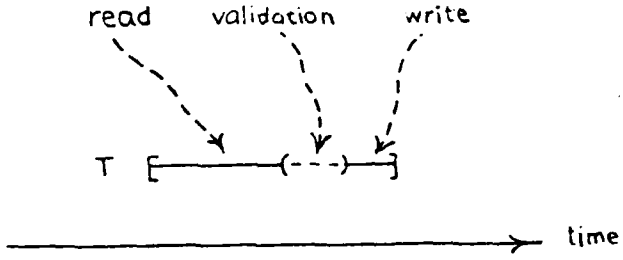


Figure 4-5: The three phases of a transaction T.

We use Fig. 4-6 to illustrate how validation works. Suppose that transaction T_2 completes its write phase by time t_4. At time t_4 transaction T_1 finishes its read phase and starts its validation. If as far as the writes of T_1 are concerned, T_1 can be thought of as if it started

after T_2 had been validated, then T_1 will be validated in our scheme. This is the case when the write set of T_2 (the set of variables T_2 writes) and the read set of T_1 (the set of variables T_1 reads) are disjoint. Assume that T_1 is successfully validated at time t_6, and T_3 finishes its read phase at time t_7. For validating T_3, the set of variables T_3 reads and the set of variables T_1 and T_2 write have to be compared. If the two sets are disjoint then T_3 can be validated. Suppose that T_3 is validated at time t_9. Then we see that the schedule corresponding to the concurrent execution of T_1, T_2 and T_3 in Fig. 4-6 is equivalent to the serial schedule which executes T_2 first and then T_1 and then T_3. This illustrates how the validation method enforces serialization.

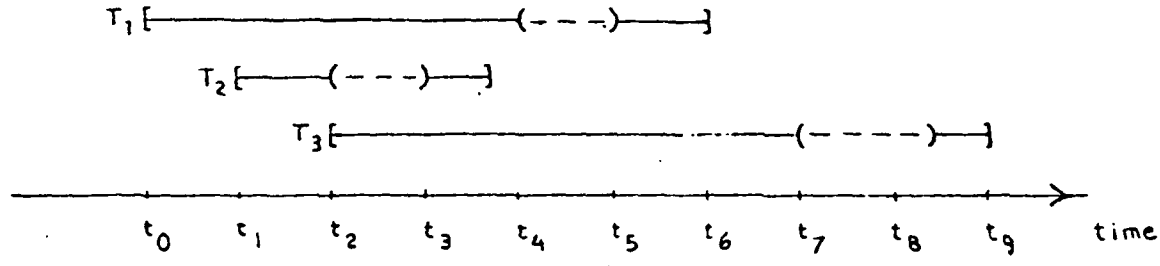


Figure 4-6: Three concurrent transactions

A straightforward implementation of the validation method is as follows. A set *W* is kept, which contains the write sets along with the validation completion times of all validated transactions. For validating a transaction T that just completed its read phase, the following steps are involved:

1. Compare the read set of T with the write sets of those transactions which are successfully validated between the start time and finish time of T.
2. If the read set is disjoint from any of those write sets examined in step (*1*) above, then do the following; otherwise restart T.
    i. Lock *W* in exclusive mode.
    ii. Compare the read set of T with the write sets of those transactions which have been successfully validated since the time T completed its read phase.
    iii. If the read set is disjoint from any of those write sets examined in step (ii) above, then validate T by performing the following operations; otherwise unlock *W* and restart T.
        a. Insert the write set of T along with the current time as the validation completion time of T into set *W*.

b. Make the local copies of T, which contain all the writes of T, global.

c. Unlock *W*.

The set *W* can be pruned down by deleting information concerning validated transactions whose validation completion times are smaller than the start time of any currently active transaction. When several transactions are ready to be validated, the main comparison step (step (*1*)), for one transaction can be carried out in parallel with the main comparison steps for other transactions on a multiprocessor. It is possible to optimize the implementation outlined above in a number of ways. We will not elaborate them here.

Validation methods are superior to locking methods for systems where transaction conflict is unlikely. Examples include query dominant systems and very large tree structured indexes. For these cases, a validation method will avoid locking overhead, and may take full advantage of a multiprocessor environment in the validation phase using the parallel validation technique presented. Some techniques are needed for determining all instances where an optimistic approach is better than a locking approach. See [Kung and Robinson 79] for more discussions on this serialization method which is not based on locking.

4.2.1.3 Remarks

Serialization methods somewhat similar to validation methods are considered in [Stearns et al. 76] for both centralized and distributed database systems. It is pointed out there that if the ordering of the equivalent serial schedule is determined on-the-fly as requests are processed, then a situation similar to deadlock may occur. The situation is called "cyclic restart", in which a finite set of transactions are caught in a loop of continually aborting and restarting each other. They solve the problem by using a preassigned ordering of transactions. The method outlined in Section 4.2.1.2, on the other hand, uses validation completion times to determine the ordering of transactions. Though the ordering is dynamic, the method is not subject to cyclic restart because in this method only validated transactions can restart other transactions. C. Papadimitriou considers the general problem of determining whether a given sequence of read and write operations corresponding to requests from several transactions is serializable [Papadimitriou 78]. He proves that the problem is NP-complete. Thus it is unlikely that there exist efficient schedulers which will recognize <u>all</u> serializable sequences of requests by the transactions. For discussions of serialization methods for distributed database systems, see, for example, [Bernstein et al. 78, Rosenkrantz et al. 78, Stonebraker 78].

4.2.2 The Approach Using Semantic Information -- A Non-Serialization Approach

As mentioned earlier, the serialization approach requires no semantic information about transactions and integrity constraints, and if such semantic information is not used, serialization is actually the only approach one can take for solving the consistency problem in a concurrent system. However, if the meanings of the transactions and integrity constraints are known *a priori*, then, as one would expect, it is often possible to design concurrent systems or algorithms enjoying high degrees of concurrency.

We assume as before that each transaction under consideration is correct if executed alone. Here we further assume that some correctness proof for each transaction is available. Such a proof must rely on and also must reflect the meanings of the transactions and the integrity constraints imposed on the database. Therefore a natural way to capture semantic information is to examine correctness proofs of the transactions.

We consider proofs using assertions [Floyd 67]. A transaction is represented as a flowchart of operations which manipulate a set of variables. Executing the transaction is viewed as moving a token on the flowchart from the input arc to an output arc. An assertion, defined in terms of the variables, is attached to each arc of the flowchart; in particular, the assertions on the input and any output arcs are the integrity constraints. A correct proof of a serial transaction amounts to demonstrating that throughout the execution of the transaction the token will always be on an arc whose assertion is true at that time, and will eventually reach an output arc. The consistency of a database under the concurrent execution of several correct serial transactions can be insured by the following scheduling policy [Lamport 76]:

> The request to execute one step in a transaction is granted only if the execution will not invalidate any of the assertions attached to those arcs where the tokens of other transactions reside at that time.

It is possible that at some time none of the transactions can be granted to execute their next steps. This "deadlock" situation can be resolved, for example, by backing up some transactions. The above scheduling policy demonstrates that at least in principle the consistency of a concurrent system can be preserved by using correctness proofs of serial transactions. In [Lamport 76], efficient schedules are derived from this scheduling policy for some concurrent systems. The schedules have the property that they preserve consistency but are not equivalent to serial schedules.

The similar idea of establishing the correctness of a concurrent system by showing that the proof of any of its sequential programs cannot be invalidated by the execution of any other program has been studied by several people, including [Ashcroft 75, Keller 76, Lamport

77, Owicki 75].

The approach of solving the consistency problem of a concurrent system by utilizing the correctness proofs of the serial transactions seems to be quite general and powerful. In this framework, with enough human ingenuity, difficult consistency problems (or their solutions) can often be solved efficiently (or explained elegantly). Some of the results in Section 4.3 below can in fact be cast in this framework. Much work remains to be done in developing mechanical ways of using this approach in designing concurrent database systems.

4.3 Algorithms for Specific Problems

When designing an asynchronous algorithm for solving a specific problem, we have control over the tasks that will be included in the algorithm. Therefore, it is possible to keep the required synchronization among the processes of an algorithm as weak as possible, by a careful design of these processes. This would not be possible for general database systems where a transaction has no idea about other transactions it might have to interact with. As a result, algorithms in this section enjoy much higher degrees of concurrency than those algorithms which are derived from the general techniques in Section 4.2.

4.3.1 Concurrent Accesses to Search Trees

We discuss how a file organized as a B-tree or a binary search tree can be accessed simultaneously by a number of processes. The goal is to insure integrity for each access while at the same time providing a high degree of concurrency and also avoiding deadlock.

Concurrent Access to B-trees

The organization of B-trees was introduced by [Bayer and McCreight 72] and some variants of it appear in [Knuth 73]. We assume that the reader is familiar with the definition of B-trees. Here we mention only that for a B-tree the leaves are all on the same level and the number of keys contained at each node except the root is between k and $2k$ for some positive integer k. The problem concerning multiple access to B-trees has been addressed in a number of papers. It appears that [Samadi 76] gave the first published solution. In his solution, exclusive locks are used by all the processes. As a search proceeds down the tree, it locks son and unlocks father until it terminates. On the other hand, an updater (insertion or deletion) locks successive nodes as it proceeds down the tree, but when a "safe" node is encountered, all the ancestors of that node are unlocked. For the insertion (or deletion) case a node is considered to be "safe" when a key can be inserted into (or removed from) that node without causing an overflow (or underflow). It is relatively easy to see that the solution preserves integrity for each access and is deadlock free.

[Bayer and Schkolnick 77] observe that when k is large the chance that an updater will cause splits or merges on nodes, especially at top levels of the tree, is small. Therefore, they propose that an updater should place weak locks such as share locks on nodes at a top section of the tree, and only later (in the second pass) convert some of these weak locks into strong locks such as exclusive locks if necessary. They present and prove the correctness of a general schema, which involves certain parameters that can be tuned to optimize the performance of the schema. Bayer and Schkolnick's solution is expected to have good average performance, especially when k is large. In the worse case however, an updater can still lock out the entire tree.

Concurrent Access to Binary Search Trees

In [Kung and Lehman 79b] algorithms for a binary search tree which can support concurrent searching, insertion, deletion and reorganization (especially, rebalancing) on the tree are proposed. In these algorithms, only writer-exclusion locks are used, simply to prevent the obvious problems created by simultaneous updates of a node by more than one process. Moreover, in these algorithms, any process locks only a small constant number of nodes at a given time, and a searcher is not blocked at all until possibly at the very end of the search when it is ready to return its answer. We discuss some general techniques that were used for achieving this high degree of concurrency.

Unlike the concurrent solutions for B-trees described above, updaters are no longer responsible for rebalancing. An update just does whatever insertion or deletion it has to do, and postpones the work of rebalancing the (possibly) unbalanced structure caused by the updating. Other processes can perform the postponed work on separate processors. Through this idea of postponement, the multiprocessing capability of a multiprocessor environment can be utilized. The same idea is used in garbage collection. Rather than performing the garbage collection itself, the deleter simply appends deleted nodes to a list of nodes to be garbage collected later. In this way, the deleter need not wait until it is safe to do the garbage collection (i.e. the time when no one else will access the deleted node), and garbage collection can be done by separate processors.

Another idea used by the algorithms is that a process makes updates only on a local copy of the relevant portion of the tree and later introduces its copy into the global tree in one step. With this technique one can get the effect of making many changes to the database in one indivisible step without having to lock a large portion of the data. However, one faces the problem of backing up processes which have read data from old copies. It turns out that because of the particular property of the tree structure, the backup problem can be handled efficiently. The copy idea is closely related to the validation method discussed in Section

4.2.1.2.

4.3.2 Asynchronous Iterative Algorithms for Solving Numerical Problems

Many numerical problems in practice are solved by iterative algorithms. For example, zeros of a function f can be approximated by the Newton iteration,

$$x_{i+1} = x_i - f'(x_i)^{-1} f(x_i),$$

and solutions of linear systems by iterations of the form,

$$X_{i+1} = AX_i + b,$$

where the X_i, b_i are n-vectors and A is an nxn matrix. In general, an iterative algorithm is defined as:

$$X_{i+1} = \varphi(X_i, X_{i-1}, \ldots, X_{i-d+1}),$$

where φ is some "iteration function". Here we are interested in parallel algorithms through which an asynchronous multiprocessor can be used efficiently to speed up the iterative process. We shall follow terminologies introduced in [Kung 76] for various classes of parallel iterative algorithms.

Iteration function φ can typically be evaluated concurrently by a number of independent processes. For example, for the Newton iteration f and f' can be evaluated concurrently, and for the matrix iteration all the components of the vector X_{i+1} can be computed simultaneously. In a straightforward synchronized (parallel) iterative algorithm, the concurrent processes that evaluate the iteration function are synchronized at each iteration step, i.e., a process is not allowed to start computing a new iterate until all the processes have finished their work for the current iterate. Thus, processes in a synchronized parallel algorithm may have to wait for each other. It has been observed that by and large iterative processes are insensitive to the ordering of evaluation as far as convergence is concerned. This observation leads to the notion of an asynchronous (parallel) iterative algorithm, in which processes are not synchronized at all. In particular, by removing the synchronization imposed on a synchronized iterative algorithm an asynchronous iterative algorithm will be obtained. In a truly asynchronous iterative algorithm, a process keeps computing new iterates by using whatever information is currently available and releases immediately its computed results to other processes. Thus, the actual iterates generated by the method depend on the relative speeds of the processes. A slightly restricted form of asynchronous iterative algorithms for solving linear systems is known as chaotic relaxation [Chazan and Miranker 69] in the literature. G. Baudet, in [Baudet 78a, Baudet 78b], reports the experimental results from the implementation of various parallel iterative algorithms on C.mmp to solve the Dirichlet problem for Laplace's equation on a rectangular two-dimensional region.

His results indicate clearly that on C.mmp asynchronous iterative methods are superior to the synchronized counterparts with respect to overall computation times. For a concise survey of parallel methods for solving equations the reader is referred to [Miranker 77].

4.3.3 Concurrent Database Reorganization

In many database organizations, the performance for accesses will gradually deteriorate due to structural changes caused by insertions and deletions. By reorganizing the database, the access costs can be reduced. The garbage collection in classical Lisp implementations can also be viewed as a database reorganization. In such an implementation when the free list is exhausted, the list processor is suspended and the garbage collector is invoked to find nodes which are no longer in use (garbage nodes) and append them to the free list. Database reorganizations are typically very time-consuming. Thus, it is desirable to reorganize a database concurrently without having to block the usual accesses to the database.

Recently there has been quite some interest in concurrent garbage collection. The goal is to collect garbage concurrently with the operations of the list processor. The first published solution is due to [Steele 75], which uses semaphore-type synchronization mechanism. [Dijkstra et al. 78] gave a solution whose synchronization is kept as weak as possible, but made no claim on the efficiency of the solution. [Kung and Song 77] gave an efficient solution by using very weak synchronization. These solutions are extremely subtle. We refer the reader to the original papers for descriptions of these solutions. Here we just discuss some experience we gained from the concurrent garbage collection problem. Contrary to what one might expect, it is <u>not</u> automatically true that because of the concurrent garbage collection the list processor will not be suspended too often and thus on the average be able to do more computations in a fixed time period. For correctness reasons, it is necessary that some synchronization overheads be introduced to the list processor, and consequently the list processor is slowed down. Also, it is inevitable that the garbage collector will sometimes perform useless work. For example, the garbage collector can be marking a set of nodes without knowing that their ancestors have just been made into garbage by the list processor. All of this affects the effectiveness of the parallel garbage collection. Similar types of performance degradation are expected in other instants of concurrent database reorganization. The central question is how to make the reorganization process effective without committing excessive synchronization costs. The problem can be extremely challenging as we have experienced in the concurrent garbage collection case. This may explain the scarcity of results available on concurrent database reorganizations today.

Memory reorganization is just one of the many "housekeeping activities" performed regularly in any large-scale computer system. Ideally, these system activities should all be

carried out by additional processors operating concurrently with the processors directly devoted to the users' computations. The system should constantly reorganize itself to improve its service to the user. The user at his end simply sees a more efficient system providing rapid responses. A feature of this approach, which is highly desirable for practical reasons, is that the speed-up can be achieved without requiring the users to rewrite their codes. It seems that concurrent reorganization (or housekeeping) represents one of the most attractive applications that asynchronous multiprocessors are capable of supporting. We expect that significant progress along this line will be made in the near future, as multiprocessors become prevalent.

4.4 Remarks for Section 4

All the efficient algorithms mentioned in Section 4.3 share a common property, namely, a process in an algorithm is never made to wait for other processes to complete their tasks. The same philosophy is used in validation methods in Section 4.2.1.2, in task scheduling [Baudet et al. 77], and in several other examples [Kung 76, Robinson 79]. This suggests that this "never-wait" principle is a useful criterion to follow in designing efficient algorithms for asynchronous multiprocessors. A typical way to achieve this goal is to use copies. After a process completes its current task, it immediately starts working on a copy of the most recent global data. Of course, validation is needed later on to determine whether or not the updated copy can be made global. Validation is not necessarily costly when it can be carried out in parallel on separate processors. Another technique to achieve the "never-wait" goal is the postponement idea as used in the concurrent binary search algorithm: a process simply ignores for the time being any work it is not allowed to perform immediately, but comes back to perform the work at a later time.

5. Concluding Remarks

One can see from the preceding sections that issues concerning algorithms for synchronous parallel computers are quite different from those for asynchronous parallel computers.

For synchronous parallel computers, one is concerned with algorithms defined on networks. Task modules of an algorithm are simply computations associated with nodes of the underlying network. Communication geometry and data movement are a major part of an algorithm. For chip implementation it is essential that the communication geometry be simple and regular, and that silicon area rather than the number of gates alone be taken into consideration. One of the important research topics in this area is the development of a new theory of algorithms that addresses issues regarding communication geometry and data movement. In particular, it would be extremely useful to have a good notation for expressing and verifying algorithms defined on networks, and to have a good complexity model for computations on silicon chips. Some initial steps along these directions have been taken by [Cohen 78, Brent and Kung 79b, Thompson 79a, Thompson 79b].

For asynchronous parallel computers, one is concerned with parallel algorithms whose task modules are executed by asynchronous processes. The major issues are the correctness and efficiency of an algorithm in the presence of the asynchronous behavior of its processes. For the general database environment where only syntactic information can be used, the serialization approach is the method for ensuring correctness. Serialization can be achieved by either locking or transaction backup. If semantic information about integrity constraints and transactions is available as in many special problem instances, then more efficient algorithms that support higher degrees of concurrency may be designed. Efficiency analysis of algorithms for asynchronous computers is usually difficult, since execution times are random variables rather than constants. Typically, techniques in order statistics and queueing models have to be employed (see, e.g., [Robinson 79]). Generally speaking, algorithms with large module granularity are well suited to asynchronous multiprocessors. In this case, a process can proceed for a long period of time before it has to wait for input from other processes. Many database applications fall into this category. Further and more detailed discussions on the programming issues raised by asynchronous multiprocessors can be found in [Newell and Robertson 75, Jones et al. 78, Jones and Schwarz 78].

References

[Aho et al. 75] Aho, A., Hopcroft, J.E. and Ullman, J.D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Massachusetts, 1975.

[Anderson and Jensen 75]
Anderson G. A. and Jensen, E. D.
Computer Interconnection Structures: Taxonomy, Characteristics, and Examples.
ACM Computing Surveys 7(4):197-213, December 1975.

[Ashcroft 75] Ashcroft, E.A.
Proving Assertions about Parallel Programs.
J. Comput. Syst. Sci. 10:110-135, January 1975.

[Barnes et al. 68] Barnes, G. H., Brown, R. M., and Kato, M., Kuck, D. J., Slotnick, D. L. and Stokes, R. A.
The ILLIAC IV Computer.
IEEE Transactions on Computers C-17(8):746-757, August 1968.

[Batcher 68] Batcher, K.E.
Sorting networks and their applications.
1968 Spring Joint Computer Conf. 32:307-314, 1968.

[Baudet 78a] Baudet, G.M.
Asynchronous Iterative Methods for Multiprocessors.
Journal of the ACM 25(2):226-244, April 1978.

[Baudet 78b] Baudet, G. M.
The Design and Analysis of Algorithms for Asynchronous Multiprocessors.
PhD thesis, Carnegie-Mellon University, Department of Computer Science, April, 1978.

[Baudet and Stevenson 78]
Baudet, G. and Stevenson, D.
Optimal Sorting Algorithms for Parallel Computers.
IEEE Transactions on Computers C-27(1):84-87, January 1978.

[Baudet et al. 77] Baudet, G., Brent, R.P. and Kung, H.T.
Parallel Execution of a Sequence of Tasks on an Asynchronous Multiprocessor.
Technical Report, Carnegie-Mellon University, Department of Computer Science, June 1977.

[Bayer and McCreight 72]
Bayer, R. and McCreight, E.
Organization and Maintenance of Large Ordered Indexes.
Acta Informatica 1(3):173-189, 1972.

[Bayer and Schkolnick 77]
Bayer, R. and Schkolnick, M.
Concurrency of Operations on B-trees.
Acta Informatica 9(1):1-21, 1977.

[Benes 65] Benes, V.E.

Mathematical Theory of Connecting Networks and Telephone Traffic. Academic Press, New York, 1965.

[Bentley and Kung 79]
Bentley, J.L. and Kung, H.T.
A Tree Machine for Searching Problems.
In *Proc. 1979 International Conference on Parallel Processing*, pages 257-266. IEEE, August, 1979.
Also Available as a CMU Computer Science Department technical report, August 1979.

[Bernstein et al. 78]
Bernstein, P.A., Goodman, N., Rothnie, J.B. and Papadimitriou, C.H.
A System of Distributed Databases (the Fully Redundant Case).
IEEE Transactions on Software Engineering SE-4:154-168, March 1978.

[Brent 74] Brent, R.P.
The Parallel Evaluation of General Arithmetic Expressions.
Journal of the ACM 21(2):201-206, April 1974.

[Brent and Kung 79a]
Brent, R.P. and Kung, H.T.
In preparation.

[Brent and Kung 79b]
Brent, R.P. and Kung, H.T.
The Area-Time Complexity of Binary Multiplication.
Technical Report, Carnegie-Mellon University, Department of Computer Science, July 1979.

[Browning 79] Browning, S.
Algorithms for the Tree Machine.
In *Introduction to VLSI Systems* by C. A. Mead and L. A. Conway, Addison-Wesley, 1979, Section 8.4.2.

[Chazan and Miranker 69]
Chazan, D. and Miranker, W.
Chaotic Relaxation.
Linear Algebra and its Applications 2:199-222, 1969.

[Chen 75] Chen, T.C.
Overlap and Pipeline Processing,
In Stone, H.S., editor, *Introduction to Computer Architecture*, pages 375-431. Science Research Associates, 1975.

[Chen et al. 78] Chen, T.C., Lum, V.Y. and Tung, C.
The Rebound Sorter: An Efficient Sort Engine for Large Files.
In *Proceedings of the 4th International Conference on Very Large Data Bases*, pages 312-318. IEEE, 1978.

[Cohen 78] Cohen, D.
Mathematical Approach to Computational Networks.
Technical Report ISI/RR-78-73, University of Southern California, Information Sciences Institute, November 1978.

[Dijkstra et al. 78] Dijkstra, E.W., Lamport, L., Martin, A.J., Shcollen, C.S. and Steffen, E.F.M.
On-the-Fly Garbage Collection: An Exercise in Cooperation.
Communications of the ACM 21(11):966-976, November 1978.

[Enslow 77] Enslow, P. H.
Multiprocessor Organization: A Survey.
ACM Computing Surveys 9(1):103-129, March 1977.

[Eswaran et al. 76] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November 1976.

[Floyd 67] Floyd, R.W.
Assigning Meanings to Programs.
In *Proc. Symposium in Applied Mathematics*, pages 19-32. American Mathematics Society, 1967.

[Flynn 66] Flynn, M. J.
Very High-Speed Computing Systems.
Proceedings of the IEEE 54(12):1901-1909, December 1966.

[Foster and Kung 79]
Foster, M. and Kung, H.T.
Design of Special Purpose VLSI Chips: Example and Opinions.
Technical Report, Carnegie-Mellon University, Department of Computer Science, September 1979.
Also appears in the *CMU Computer Science Research Review 1978-79.*

[Fuller, et al. 77] Fuller, S. H., Jones, A. K. and Durham, I. (Eds.).
*The Cm** *Review Report.*
Technical Report, Carnegie-Mellon University, Department of Computer Science, June 1977.

[Gray 78] Gray, J.
Notes on Data Base Operating Systems.
In *Lecture Notes in Computer Science 60: Operating Systems*, pages 393-481. Springer-Verlag, Berlin, Germany, February, 1978.

[Guibas et al. 79] Guibas, L.J., Kung, H.T. and Thompson, C.D.
Direct VLSI Implementation of Combinatorial Algorithms.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design,* Fabrication, California Institute of Technology, January, 1979.

[Hallin and Flynn 72]
Hallin, T.G. and Flynn, M.J.
Pipelining of Arithmetic Functions.
IEEE Transactions on Computers C-21:880-886, 1972.

[Heart et al. 73] Heart, F. E., Ornstein, S. M., Crowther, W. R. and Barker, W. B.
A New Minicomputer/Multiprocessor for the ARPA Network.
In *AFIPS Conference Proceedings, NCC '73*, pages 529-537. AFIPS, 1973.

[Heller 78] Heller, D.
A Survey of Parallel Algorithms in Numerical Linear Algebra.
SIAM Review 20(4):740-777, October 1978.

[Jones and Schwarz 78]

Jones, A.K. and Schwarz, P.
A Status Report: Experience Using Multiprocessor Systems.
Manuscript, CMU Computer Science Department.
To appear.

[Jones et al. 78] Jones, A.K., Chansler, R.J. Jr., Durham, I., Feiler, P.H., Scelza, D.A., Schwans, K. and Vegdahl, S.R.
Programming Issues Raised by a Multiprocessor.
Proceedings of the IEEE 66(2):229-237, February 1978.

[Karp 72] Karp, R. M.
Reducibility Among Combinational Problems,
Complexity of Computer Computations, pages 85-104. Plenum Press, New York, 1972.

[Kautz et al. 68] Kautz, W.H., Levitt, K.N. and Waksman, A.
Cellular Interconnection Arrays.
IEEE Transactions on Computers C-17(5):443-451, May 1968.

[Keller 76] Keller, R.M.
Formal Verification of Parallel Programs.
Communications of the ACM 19:371-384, July 1976.

[Knuth 73] Knuth, D. E.
The Art of Computer Programming. Volume 3: *Sorting and Searching.*
Addison-Wesley, Reading, Massachusetts, 1973.

[Kosaraju 75] Kosaraju, S.R.
Speed of Recognition of Context-Free Languages by Array Automata.
SIAM J. on Computing 4:331-340, 1975.

[Kuck 77] Kuck, D. J.
A Survey of Parallel Machine Organization and Programming.
ACM Computing Surveys 9(1):29-59, March 1977.

[Kuck 78] Kuck, D. J.
The Structure of Computers and Computations.
John Wiley and Sons, New York, 1978.

[Kuck 68] Kuck, D.J.
ILLIAC IV Software and Application Programming.
IEEE Transactons on Computers C-17:758-770, 1968.

[Kung 76] Kung, H. T.
Synchronized and Asynchronous Parallel Algorithms for Multiprocessors,
In Traub, J. F., editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 153-200. Academic Press, New York, 1976.

[Kung 79] Kung, H.T.
Let's Design Algorithms for VLSI Systems.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design,* Fabrication, California Institute of Technology, January, 1979.

[Kung and Lehman 79a]
Kung, H.T. and Lehman, P.L.

VLSI Algorithms for Relational Databases.
To appear.

[Kung and Lehman 79b]
Kung, H.T. and Lehman, P.L.

A Concurrent Database Problem: Binary Search Trees.
Technical Report, Carnegie-Mellon University, Department of Computer Science, September 1979.
An abstract appears in the *Proc. of the Fourth International Conference on Very Large Databases.* The full paper is to appear in *ACM Transactions on Database Systems.*

[Kung and Leiserson 79]
Kung, H.T. and Leiserson, C.E.
Systolic Arrays (for VLSI).
In Duff, I. S. and Stewart, G. W., editor, *Sparse Matrix Proceedings 1978*, pages 256-282. Society for Industrial and Applied Mathematics, 1979.
A slightly different version appears in *Introduction to VLSI Systems* by C. A. Mead and L. A. Conway, Addison-Wesley, 1979, Section 8.3.

[Kung and Leiserson 78]
Kung, H.T. and Leiserson, C.E.
Systolic Array Apparatuses for Matrix Computations.
U.S. Patent Application, Filed December 11, 1978.

[Kung and Papadimitriou 79]
Kung, H.T. and Papadimitriou, C.H.
An Optimality Theory of Concurrency Control for Databases.
In *Proc. ACM-SIGMOD 1979 International Conference on Management of* Data, pages 116-126. ACM, May, 1979.

[Kung and Robinson 79]
Kung, H.T. and Robinson, J.T.
On Optimistic Methods for Concurrency Control.
Technical Report, Carnegie-Mellon University, Department of Computer Science, September 1979.
An abrstract appears in the *Proc. Fifth International Conference on Very Large Data Bases*, October 1979. The full paper is to appear in *ACM Transactions on Database Systems.*

[Kung and Song 77]
Kung, H.T. and Song, S.W.
A Parallel Garbage Collection Algorithm and Its Correctness Proof.
In *Proc. Eighteenth Annual Symposium on Foundations of Computer* Science, pages 120-131. IEEE, October, 1977.
A revised version is to appear in *Communications of the ACM.*

[Kung and Stevenson 77]
Kung, H.T. and Stevenson, D.
A Software Technique for Reducing the Routing Time on a Parallel Computer with a Fixed Interconnection Network,
In Kuck, D. J., Lawrie, D.H. and Sameh, A.H., editor, *High Speed Computer and Algorithm Organization*, pages 423-433. Academic Press, New York, 1977.

[Lamport 76] Lamport, L.

Towards a Theory of Correctness for Multi-user Data Base Systems.
Technical Report CA-7610-0712, Massachusetts Computer Associates, Inc., October 1976.

[Lamport 77] Lamport, L.
Proving the Correctness of Multiprocess Programs.
IEEE Transactions on Software Engineering SE-3(2):125-143, March 1977.

[Lawrie 75] Lawrie, D.H.
Access and Alignment of Data in an Array Processor.
IEEE Transactions on Computers C-25(12):1145-1155, December 1975.

[Leiserson 79] Leiserson, C.E.
Systolic Priority Queues.
In *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, January, 1979.
Also available as a CMU Computer Science Department technical report, April 1979.

[Levitt and Kautz 72]
Levitt, K.N. and Kautz, W.H.
Cellular Arrays for the Solution of Graph Problems.
Communications of the ACM 15(9):789-801, September 1972.

[Mead and Conway 79]
Mead, C.A. and Conway, L.A.
Introduction to VLSI Systems.
Addison-Wesley, Reading, Massachusetts, 1979.

[Mead and Rem 79]
Mead, C.A. and Rem, M.
Cost and Performance of VLSI Computing Structures.
IEEE Journal of Solid State Circuits SC-14(2):455-462, April 1979.

[Miranker 71] Miranker, W.L.
A Survey of Parallelism in Numerical Analysis.
SIAM Review 13:524-547, 1971.

[Miranker 77] Miranker, W.L.
Parallel Methods for Solving Equations.
Technical Report RC6545, IBM T.J. Watson Research Center, May 1977.

[Mukhopadhyay and Ichikawa 72]
Mukhopadhyay, A. and Ichikawa, T.
An n-Step Parallel Sorting Machine.
Technical Report 72-03, The University of Iowa, Department of Computer Science, 1972.

[Newell and Robertson 75]
Newell, A. and Robertson, G.
Some Issues in Programming Multiprocessors.
Behavior Research Methods and Instrumentation 7(2):75-76, March 1975.

[Oleinick 78] Oleinick, P.N.

The Implementation and Evaluation of Parallel Algorithms on C.mmp.
Technical Report, Carnegie-Mellon University, Department of Computer Science, November 1978.

[Owicki 75] Owicki, S.
Axiomatic Proof Techniques for Parallel Program.
PhD thesis, Cornell University, Department of Computer Science, 1975.

[Papadimitriou 78] Papadimitriou, C.H.
Serializability of Concurrent Updates.
Harvard University.
To appear in JACM.

[Pease 68] Pease, M.C.
An Adaptation of the Fast Fourier Transform for Parallel Processing.
Journal of the ACM 15:252-264, April 1968.

[Peleg and Rosenfeld 78]
Peleg, S. and Rosenfeld, A.
Determining Compatibility Coefficients for Curve Enhancement Relaxation Processes.
IEEE Transactions on systems, Man, and Cybernetics SMC-8(7):548:556, July 1978.

[Ramamoorthy and Li 77]
Ramamoorthy, C.V. and Li, H.F.
Pipeline Architecture.
Computing Surveys 9(1):61-102, March 1977.

[Robinson 79] Robinson, J.T.
Some Analysis Techniques for Asynchronous Multiprocessor Algorithms.
IEEE Transactions on Software Engineering SE-5(1):24-31, January 1979.

[Rosenkrantz et al. 78]
Rosenkrantz, D.J., Stearns, R.E. and Lewis II, P.M.
System Level Concurrency Control for Distributed Database Systems.
ACM Transactions on Database Systems 3(2):78-198, June 1978.

[Samadi 76] Samadi, B.
B-trees in a System with Multiple Users.
Information Processing Letters 5(4):107-112, October 1976.

[Sameh 77] Sameh, A.H.
Numerical Parallel Algorithms -- A Survey.
In *High Speed Computer and Algorithm Organization*, pages 207-228. Academic Press, New York, 1977.

[Smith 71] Smith III, A.R.
Two-Dimensional Formal Languages and Pattern Recognition by Cellular Automata.
In *Proc. 12th IEEE Symposium on Switching and Automata Theory*, pages 144-152. IEEE, 1971.

[Stearns et al. 76] Stearns, R.E., Lewis , P.M. II and Rosenkrantz, D.J.
Concurrency Control for Database Systems.

In *Proc. Seventh Annual Symposium on Foundations of Computer Science*, pages 19-32. IEEE, 1976.

[Steele 75] Steele, G.L., Jr.
Multiprocessing Compactifying Garbage Collection.
Communications of the ACM 18(9):125-143, September 1975.

[Stone 75] Stone, H.S.
Parallel Computation.
In Stone, H.S., editor, *Introduction to Computer Architecture*, pages 318-374. Science Research Associate, Chicago, 1975.

[Stone 71] Stone, H.S.
Parallel Processing with the Perfect Shuffle.
IEEE Transactions on Computers C-20:153-161, February 1971.

[Stonebraker 78] Stonebraker, M.
Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES.
In *Proc. Third Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 235-258. Lawrence Berkeley Laboratory, University of California, Berkeley, August, 1978.

[Sutherland and Mead 77]
Sutherland, I.E. and Mead, C.A.
Microelectronics and Computer Science.
Scientific American 237:210-228, 1977.

[Thompson and Kung 77]
Thompson, C.D. and Kung, H.T.
Sorting on a Mesh-Connected Parallel Computer.
Communications of the ACM 20(4):263-271, April 1977.

[Thompson 79a] Thompson, C.D.
Area-Time Complexity for VLSI.
In *Proc. Eleventh Annual ACM Symposium on Theory of Computing*, pages 81-88. ACM, May, 1979.

[Thompson 79b] Thompson, C.D.
A Complexity Theory for VLSI.
PhD thesis, Carnegie-Mellon University, Department of Computer Science, 1979.

[Voigt 77] Voigt, R.G.
The Influence of Vector Computer Architecture on Numerical Algorithms,
In Kuck, D.J., Lawrie, D.H. and Sameh, A.H., editors, *High Speed Computer and Algorithm Organization*, pages 229-244. Academic Press, New York, 1977.

[Von Neumann 66] Von Neumann, J.
Theory of Self-Reproducing Automata.
University of Illinois Press, Urbana, Illinois, 1966.

[Wulf and Bell 72] Wulf, W. A. and Bell, C. G.
C.mmp -- A Multi-Mini-Processor.
Proceedings Fall Joint Computer Conference 41:765-777, 1972.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (14) CMU-CS-79-143	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE STRUCTURE OF PARALLEL ALGORITHMS	5. TYPE OF REPORT & PERIOD COVERED (9) Interim rept. / 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) H. T. Kung	8. CONTRACT OR GRANT NUMBER(s) (15) N00014-76-C-0370 / NSF-MCS78-236?6
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213 (12) 39	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217	(11) 12. REPORT DATE Aug[illegible]979 / 13. NUMBER OF PAGES 58
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED 403087

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)